MASTER

Multi-standard multi-channel channel decoder architecture for mobile applications

Tong, W.

Award date:
2009

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computer Science

MASTER'S THESIS

Multi-Standard Multi-Channel Channel Decoder Architecture for Mobile Applications

by
W. Tong

Document type:	MSc thesis Eindhoven University of Technology (TU/e)
MSc work:	Performed at ST-Ericsson, Technology & Tools, Advanced R&D, HTC Eindhoven
Period of work:	Nov 1, 2008 – Aug 18, 2009
Supervisor:	Prof. Dr. C. H. van Berkel
Tutor:	MSc R.J.M. Nas

Multi-Standard Multi-Channel Channel Decoder Architecture for Mobile Applications

Master's Thesis

Abstract

In the context of software defined radio (SDR), multiple radios can run simultaneously on a shared hardware platform. Flexible Outer Receiver Architecture (FLORA), as a configurable hardware accelerator of this hardware platform, was designed to handle channel decoding jobs for multiple radios. Each channel decoding job has its own execution rate and can be started or stopped independently from other jobs by the user. This thesis presents a design flow to schedule these channel decoding jobs running on FLORA. We show a compile-time scheduling which can find a hardware partition of FLORA and group several fine-granularity tasks to a coarse-granularity task. We also propose a run-time scheduling and resource management which can handle dynamic combinations of the jobs. We and illustrate the usage of these techniques and calculate schedulability for the combination of (resource models of) DVB-T, DVB-SH, and LTE channel decoding jobs running on a simulation platform.

Legal Information

Disclaimer

The contents of this document are subject to change without prior notice. ST-Ericsson makes no representation or warranty of any nature whatsoever (neither expressed nor implied) with respect to the matters addressed in this document, including but not limited to warranties of merchantability or fitness for a particular purpose, interpretability or interoperability or, against infringement of third party intellectual property rights, and in no event shall ST-Ericsson be liable to any party for any direct, indirect, incidental and or consequential damages and or loss whatsoever (including but not limited to monetary losses or loss of data), that might arise from the use of this document or the information in it.

ST-Ericsson and the ST-Ericsson logo are trademarks of the ST-Ericsson group of companies or used under a license from STMicroelectronics NV or Telefonaktiebolaget LM Ericsson.

All other names are the property of their respective owners.

Contents

Preface

This document is my Master's thesis and is the result of my graduation project to obtain the degree of Master of Science with a specialization in Embedded Systems. The project was started from November 2008 and finished in August 2009. It was carried out in Advanced R&D group at ST-Ericsson in Eindhoven.

I would like to express my gratitude to Prof. Kees van Berkel for his excellent guidance and for giving me the opportunity to carry out my research within SDR project in ST-Ericsson. I would like to thank Rick Nas, who spent many hours to help me understand FLORA and SDR project. He guided me through all the stages of the research work and thesis writing. I also like to thank Orlando Moreira for his help with scheduling and the SDF graph related issues. Without the discussion with him, the result would not be possible. Finally, I greatly appreciate the support from my family and my girl friend during past two years. Thanks a lot!

Wei Tong

Eindhoven, August 2009

1 Introduction

Nowadays, wireless communication is becoming more and more important in our life. Various wireless standards, such as 3G standards, GPS, Bluetooth, WIFI, and mobile TV standards like Digital Video Broadcasting-satellite services to handhelds (DVB-SH), have been developed to provide the end users with better services and experiences. In the near future, Long-term evolution (LTE) and ultra-wideband (UWB) are also coming. The wireless standards are developing and evolving rapidly.

For the cell phone manufacturers, to design the chipset for each standard and to react rapidly to market requirements have become very challenging. Due to limited battery capacity of the handsets and the intensive computational workloads in wireless communication, application specific integrated circuits (ASIC) are often used to carry out the algorithms. ASICs are hardwired and with limited flexibility. Therefore, manufacturers have to design different ASICs for various standards. But at present, the life cycle of the standard is becoming shorter and shorter. Even within this shorter life cycle, the algorithms of the standard are still evolving. Poor programmability and configurability of the ASIC force the manufacturers to redesign the chips if they want to use the new algorithms or design the solution for a new standard.

To meet the demand of the seamless communication between various networks and to reduce the cost of designing mobile platform, the software defined radio (SDR) is proposed. It is defined as: "*Radio in which some or all of the physical layer functions are software defined*" in [6].

In our project, we mainly focus on SDR baseband processing out of the whole physical layer. Our ambition is to process multi-standard radio baseband decoding on a shared hardware platform. In order to achieve the goal, a heterogeneous multiprocessor system on chip (MPSoC) platform is designed with a balance among flexibility, power consumption and computational power. As a configurable hardware accelerator, Flexible Outer Receiver Architecture (FLORA) is one of the subsystems in this platform. FLORA consists of several configurable subsystems such as *Viterbi*, *Turbo,* and *De-interleaver,* to handle the channel decoding for different radio standards. To efficiently make use of the multi-standard multi-channel decoding capability of FLORA, a smart scheduling strategy is desired to guarantee the real-time performance of the running radio standards. This thesis proposes a design flow which enables us to non-preemptively schedule multi-radio applications with a dynamic job-mix on FLORA.

1.1 Overview of Baseband Processing

As shown in Figure 1-1, baseband processing typically consists of 3 stages: digital front end (DFE), modem and codec.

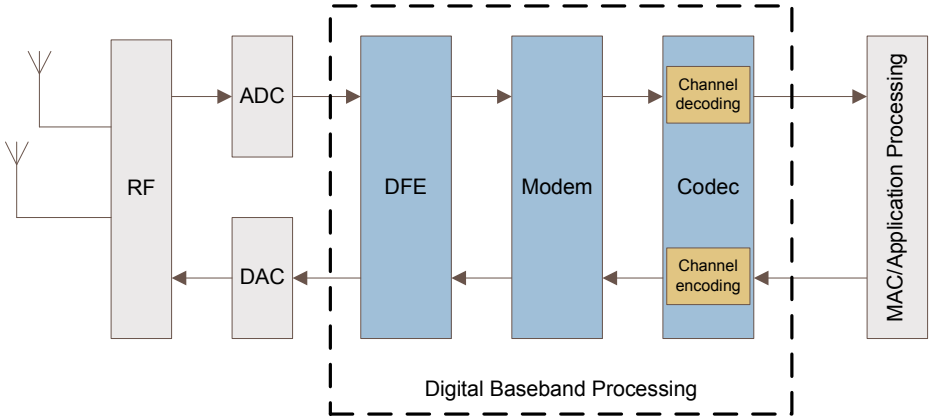

Figure 1-1 The baseband processing flow

1.1.1 DFE stage

As an enhancement for analog front end (AFE), DFE accounts for the major part of the re-configurability of the transceiver. By means of various filters, 3 essential functions are achieved in DEF stage [1][2].

- IQ transposition. Convert the digitalized real signal to complex signal and vice versa.
- Sample rate conversion (SRC). Convert the digitalization rate to the rate that fits the current standard.
- Channel selection. Select the proper channel. It includes conversion to baseband and channel filtering.

Because of the high computational load and similarity among the algorithms of different wireless standards, DFE is normally implemented in terms of a configurable hardware but not mapped to a fully programmable processor.

1.1.2 Modem stage

The modem, also called "inner transceiver", performs several functions such as modulation, demodulation, mapping, de-mapping, channel estimation, channel equalization, and so on.

The modem stage gains most from the flexibility in the hardware. This is because:

- The standards are highly diverse and algorithms are rather complex.
- The functions in modem stage involve intensive computational load and convolution based operations, for instance, FFT and correlation, which can be efficiently implemented on vector processor.

- The standard leaves the freedom to manufacturers to design their own algorithms with better performance, which can be a differentiator from the others. Also, with the evolvement of the standards, manufacturers need to adapt algorithms to achieve better performance.

Lots of researches have been done both in software and hardware related with modem stage in SDR baseband processing.

1.1.3 Codec stage

The codec stage, also called "outer transceiver", performs bit-based operation such as *(de)interleaving, (de)puncturing*, and *channel decoding/encoding* (*Turbo, Viterbi, ReedSolomon* etc.). As seen from Figure 1-1, codec stage can be split into 2 parts according to the direction of communication. If we take downlink in mobile communication as an example, in the transmitter (base station) side, redundancy data is added by means of channel encoding. This enables error correction at the receiver (mobile) side to improve the reliability of the transmission.

Since only a limited number of functions are required to support multi-standards and these functions are determined by the standards with little flexibility left to manufacturers, a hardware accelerator with a medium degree of reconfigurability will be sufficient.

1.2 The FLORA Architecture

In SDR enabled terminals, baseband processing is mapped on a programmable hardware platform. Such a multi-standard hardware platform is proposed in [3], as shown in Figure 1-2.

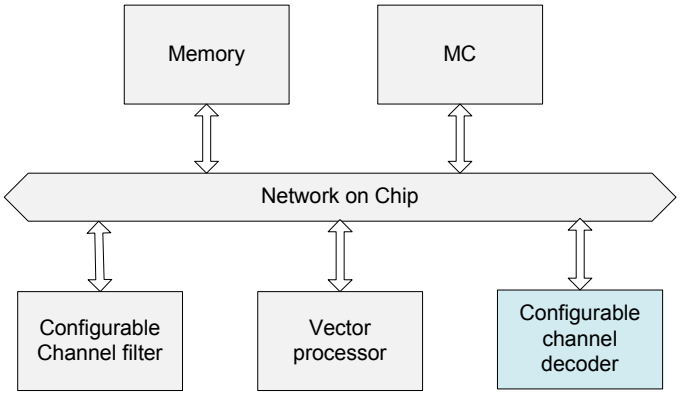

Figure 1-2 Hardware for SDR baseband

A microcontroller (MC) is used to control baseband tasks. The DFE stage can be mapped on the configurable channel filter. One or several vector processors can handle the multi-standard modem stages efficiently. And a reconfigurable hardware accelerator accounts for multi-standard channel decoding (error correction) in codec stage.

In our approach, FLORA is designed to be such a reconfigurable channel decoder. Figure 1-3 depicts the hardware architecture of FLORA.

The *De-interleaver* is normally the first part of the decoding chain. It can read the data from external memory and forward the data to the next engine as a regular DMA. It can also perform permutation by programming an integrated address generator, which is a small vector processor. The vector processor in *De-interleaver* is fully programmable.

The *Depuncture* unit is needed when convolutional codes are involved in the broadcast standard (turbo, viterbi). It has several puncturing patterns that result in different code rates (CR). The *Depuncture* unit in FLORA is a generic unit that is compliant to several standards. It is double buffered at input as well as output resulting in non-stalling dataflow.

The *Viterbi* decoder and the *ReedSolomon* decoder are also designed for multiple standards and both have small input/output buffer. Unlike the *Viterbi* decoder and the *ReedSolomon* decoder, the *Turbo* decoder have a big input and output buffer. Therefore, it can work in block mode meaning that it operate in a block based fashion. Each execution will consume and produce a block of data. The *Turbo* decoder in FLORA is configurable. By setting different parameters, it can handle different standards.

The *Viterbi, ReedSolomon and Turbo* decoders perform real decoding functions, which remove the redundancy and convert transmission data to user data.

The *Descrambler* is a multi standard engine compliant to several standards. Every energy dispersal polynomial can be programmed to a max degree of 31. In one clock cycle it can generate aone byte of output data.

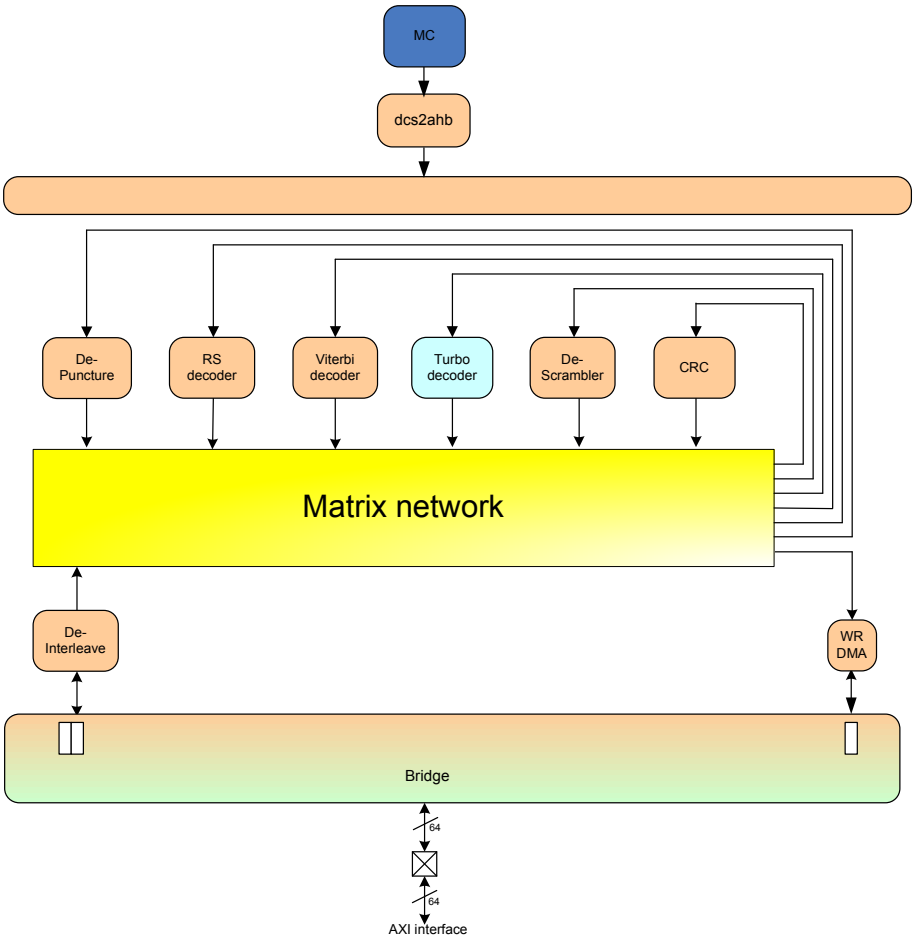

Figure 1-3 The FLORA architecture

The hardware units inside FLORA are connected to a matrix network. The routing of the connections is decided by the applications mapped on it. The routing will chain several hardware units together, and they can work simultaneously to execute one or more applications.

All the hardware units in the FLORA are controlled and configured by an MC. In the low power context, such as the mobile platform, MC is normally an ARM processor. If some of the hardware units in FLORA need to decode another standard, the ARM is responsible for configuring the parameters of the hardware units in FLORA via Advanced Peripheral Bus (APB) [7]. MC also accounts for the scheduling of the tasks running on the hardware units. The start time and execution order of the tasks are all decided by the ARM. The input data of FLORA is from the outside. A multi layer Advanced eXtensible Interface (AXI) bus [7] glues the FLORA with the rest of baseband processing platform.

1.3 Computation Model

In the previous section, we introduce the baseband processing and the FLORA architecture for the outer receiver. In this section, we discuss how to model the applications that are mapped on FLORA.

In SDR baseband processing, several applications can be executed simultaneously. The channel decoding parts of these applications are mapped on FLORA. The channel decoding parts are called jobs in this thesis. One job may consist of several tasks. In the context of SDR baseband, these jobs are normally hard real-time jobs [11]. This means that all the jobs must meet their throughput/latency requirements at all times. The guarantee has to be made to avoid the unexpected events. Therefore, firstly, the temporal behavior of the jobs must be predictable. Secondly, the resource consumption of each job must be well known, to arrive at an efficient and correct mapping.

Due to these purposes, a job model is needed to enable the analysis of the scheduling and resource management during the compile time. The job model needs to be abstracted to a certain level, such that we can do a quick simulation and verification in the early phase of the design, without knowing any unnecessary details. But the model is also required to keep proper temporal behavior and reflect resources consumption. A task graph is a good way to model these jobs. It includes the computational tasks in the job, and with edges between tasks to depict the data flow and dependencies. Synchronous Dataflow Graphs (SDF) [8] is a sub set of task graphs. It offers design-time predictability. In section 1.3.1, SDF is discussed in detail. Next, in section 1.3.2, we give the detail of homogeneous synchronous dataflow graph (HSDF), a special case of SDF.

1.3.1 Synchronous Dataflow Graph

As a special case of dataflow graphs, the SDF model was introduced in 1987 by Edward A. Lee and David G. Messerschmitt [8]. The definition of SDF was given in [9], as shown below:

Definition 1(SDF Graph): An SDF graph is defined by the tuple (V, E, d, P, I, O) where

- V is a set of actors (vertices/nodes of the graph),
- $E \subseteq V \times V$, is a set of directed edges,
- d: $E \rightarrow \mathbb{N}$ is a function that specifies the number of initial tokens (delay) on an $edge(u,v) \in E$,
- P: $V \rightarrow \mathbb{N}$ is a function that defines the execution time of an actor $v \in V$. $P(v)$ is always a constant number during the execution of SDF graph,
- I : $E \rightarrow \mathbb{N}$ is a function that describes the number of tokens consumed by an actor on $edge(u,v) \in E$,
- O: $E \rightarrow \mathbb{N}$ is a function that describes the number of tokens produced by an actor on $edge(u,v) \in E$.

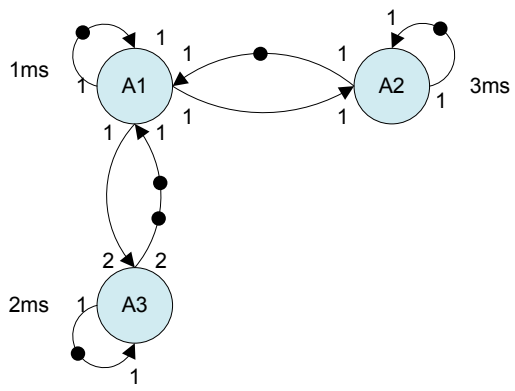

Figure 1-4 SDF graph example

Figure 1-4 shows an example of SDF graph. The vertices/nodes in the graph are called actors, which correspond to the tasks in a job. The actor consumes a certain number of input tokens and produces output tokens. The number of input or output tokens for each firing is specified next to the head or tail of the edges. An actor is always enabled when the specified number of tokens is available on all of its input edges. Once it is enabled, it can fire. A worst cast execution time (WCET) is normally used to annotate the firing time of the actor. WCET is defined as the maximum length of time the task could take to execute on a specific hardware platform. In the illustrations of the SDF graph above, A1 takes 1ms to finish an execution. Once the firing is done, all the input tokens of the actor are consumed, and all the output tokens are produced.

Edges in the SDF graph represent the data dependences between the actors. They can be seen as channels or infinite-sized FIFOs, which carry the tokens flowing from an actor to another one. Edge may have a number of initial tokens, sometimes called its delay, which are depicted as bullets on the edge. Initial tokens on back edge are often used to specify the FIFO size between actors. For instance, in the Figure 1-4, the initial tokens between A3 and A1 can represent that FIFO size between them is 2. Maximally, A1 can fire twice before A3 finishes its first execution. A self-edge of an actor means that the actor can't fire again until the previous execution is done. It is used to prevent the concurrent firings of same actor and can also represent the state of the actor between firings.

1.3.2 Homogenous Synchronous Dataflow Graph

HSDF graph is a special case of SDF graph. It is in a more restricted form than normal SDF graph: the execution of every actor in HSDF graph consumes exactly 1 input token and produces 1 output token. The definition of HSDF graph is given below.

Definition 2 (HSDF Graph): An HSDF graph is defined by the tuple (V, E, d, P), where:

- V is a set of actors (vertices/nodes of the graph),
- $E \subseteq V \times V$ is a set of directed edges,
- d: $E \rightarrow \mathbb{N}$ is a function that specifies the number of initial tokens (delay) on an $edge(u,v) \in E$,

- P: $V \rightarrow \mathbb{N}$ is a function that defines the execution time of an actor $v \in V$. $P(v)$ is always a constant number during the execution of HSDF graph.

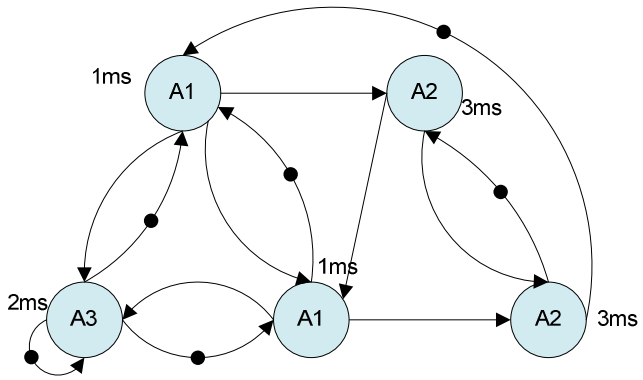

Figure 1-5 HSDF graph example

Any SDF graph can be transformed to a HSDF equivalent graph [10]. Figure 1-5 is the HSDF graph after the transformation performed on the SDF graph in Figure 1-4. The reason we need this transformation is that there are many techniques in HSDF graph that can enable throughput, latency, and scheduling analysis.

The self-timed execution of the HSDF graph is an execution in which every actor is fired as soon as it is enabled. We make the conservative assumptions: input data and output space must be available at the beginning of the firing and input space and output tokens are released at the end of the firing. The self-timed execution of HSDF graph reflects the task-level parallelism. The Figure 1-6 shows the self-timed execution of the HSDF graph in Figure 1-5.

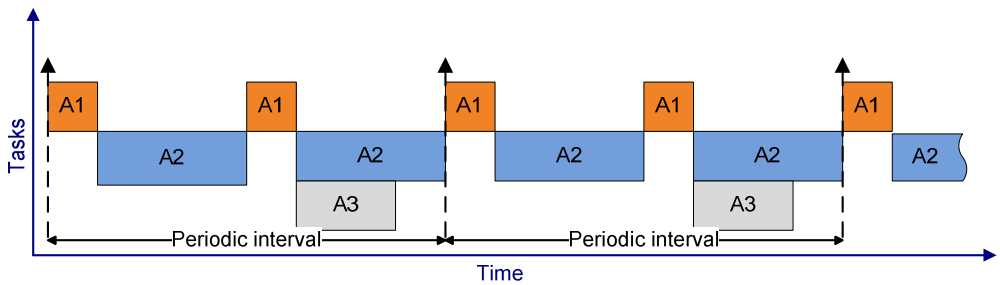

Figure 1-6 self-timed execution of the example in Figure 3-2

With the time elapsing, as seen from the Figure 1-6, the example in Figure 1-5 enters a periodic regime. This is not a single special case, but a general property of self-timed executed HSDF graph. It indicates that, for every HSDF graph (or SDF graph, since all SDF graphs can be transformed to HSDF graphs), after the transient phase, it will always reach a steady state. This steady state is repeating itself with a certain period, which is found to be equal to integral multiple of the so-called Maximum Cycle Mean (MCM). Before we introduce the MCM, we first give the definition of Cycle Mean (CM) [11].

Definition 3 (CM): The cycle mean of a cycle c in a HSDF graph $G=(V,E,d,P)$ is defined as,

$$CM(c)=\frac{\sum_{a\in Vs(c)}P(a)}{\sum_{e\in Es(c)}d(e)} \qquad 1.1$$

where

- C is a set of directed cycles in HSDF graph G. $\forall c\in C$, is a cycle directed from an actor to itself, and transverses each node in it once.
- Vs: $C\rightarrow V$ is a function that specifies all the actors in the directed cycle $c\in C$. $Vs(c)\subseteq V$,
- Es: $C\rightarrow E$ is a function that specifies all the edges in the directed cycle $c\in C$. $Es(c)\subseteq E$.

Based on the definition of CM, the definition of MCM is given below.

Definition 4 (MCM): the Maximum Cycle Mean of HSDF graph $G=(V,E,d,P)$ is defined as,

$$MCM(G)=\max_{c\in C_G}CM(c) \qquad 1.2$$

where C is a set of directed cycles in HSDF graph G.

The guaranteed minimum throughput of the HSDF graph is the inverse of MCM. Thus, if we use HSDF graph to model a hard real-time job, the throughput can be analyzed by calculating MCM.

The equation 1.2 shows a straightforward way of calculating MCM. And many polynomial algorithms are created to find MCM in a HSDF graph [13]. The MCM in Figure 1-5 is 8ms, same as periodic interval in Figure 1-6. The throughput of this example is $\frac{1}{MCM}=125$.

Besides throughput requirements, latency is another form of constraint of hard real-time jobs. Latency constraints can be modeled by the method proposed in [12]. Therefore, in this thesis, we will only talk about throughput constraints.

In this thesis, all the SDF and HSDF graphs are in self-timed execution. The actors will fire immediately once they are enabled.

1.4 Organization of Thesis

The rest of the thesis is organized as follows. In chapter 2, we describe the problems we want to address and the goals we want to achieve. In chapter 3, we motivate and describe our scheduling approach. In chapter 4 we provide some detailed information related with our implementation and simulation. In chapter 5, we present our result. Chapter 6 concludes the thesis and discusses the future work.

2 Problem Statement

2.1 Challenges of Scheduling FLORA

FLORA as a configurable hardware accelerator can be seen as a restricted multiprocessor platform, on which channel decoding part of the radio can be mapped. There are some hardware-constraints that make the scheduling of the tasks on FLORA more challenging. The main challenges are specified as follows:

- Limited buffer size. The connections between hardware units in FLORA are implemented in hardware buffer. Due to the expensive cost of on-chip memory, the input/output buffer size of each hardware unit in FLORA just meets or slight exceeds minimum functional requirement. Due to the strict buffer size constraint, it is impossible to store extra data during the processing. The flexibility of scheduling is limited because of the restricted buffering.
- Scheduling overhead. As we mentioned before, FLORA is controlled by MC via control bus. All the configuration and scheduling commands for FLORA are sent by MC. If MC treats every hardware unit as a scheduling unit, there will be a nontrivial scheduling workload for MC. Moreover, if we schedule the operations of hardware units in a fine granularity (i.e. bit, byte level), MC has to continuously configure and schedule hardware units. This will cause awful amount of communication workload. It is desired to find a scheduling strategy that enables us to schedule several hardware units as a single unit in coarse granularity (i.e. thousands of bits, bytes).
- Dynamic combination of jobs. In the context of SDR, the end users can start/stop any radio applications, such as LTE, WIFI, DVB-T, DVB-SH, at any time. There are various combinations of these applications during run time. Purely static scheduling is not a good choice here. A proper scheduling method is needed to handle dynamic combination of the jobs in run time with the guarantee of hard real-time performance.

2.2 Objectives

This thesis is focusing on overcoming these scheduling challenges for FLORA. In the end, the following results will be delivered.

- The simulation platform of FLORA will be created using Coware ESL and Virtual Platform Unit (VPU) technology.
- The behavioral models of different radio applications will be created in Coware. These applications include DVB-SH, DVB-T, and LTE.

- An online scheduling approach that fits well with the rest of system will be designed and implemented in Coware.
- The scheduling simulations of the different radio applications as well as dynamic combination of them will be done using Coware, based on various mapping of different radio models.
- Finally, a document about the approach, implementation, and simulation results will be written.

3 Overall Approach

3.1 Introduction

The problem we want to address in this chapter is finding a scheduling strategy which can handle a dynamic combination of several hard-real-time jobs running on FLORA. A lot of research has been done related with scheduling hard-real-time jobs on a multiprocessor and some achievements have been made. The scheduling strategy proposed in [11] addresses the problem of how to schedule a dynamic mix of hard-real-time jobs on a heterogeneous multiprocessor. The hardware model used in [11] is under the assumption that there are enough buffers for communication and the system is preemptible. But this is not the case for FLORA. FLORA can't be easily modeled as an instance of the multiprocessor system template in [9], simply because that there are too many hardware constraints in FLORA. These constraints such as limited buffer size, non preemptible hardware, make our design more challenging. In order to overcome these hardware constraints, we propose a design flow shown in Figure 3-1 and Figure 3-2.

Our approach consists of 2 parts: compile-time scheduling and run-time scheduling. There are several advantages of doing partial scheduling during compile time. Firstly, there are only a limited number of radio applications that will be mapped on FLORA and we have the knowledge about these radios at compile time. Based on that knowledge, some scheduling decisions can be made in advance. Secondly, there is no time limitation during the compile time, so it is possible to reduce the run-time scheduling workload by using complex algorithms and methods.

From the specifications of the radio standards, the implementation independent SDF graphs can be derived to describe the temporal behavior of radios. This kind of SDF graph is not related with the hardware at all. However, once we want to analyze the temporal behaviors of the radios that are mapped onto FLORA, we need to create the implementation aware SDF graphs where the hardware constraints, such as buffer size and processing power are taken into account. We refer to the method that derives the implementation aware SDF graph from the implementation independent SDF graph and hardware constraints as *execution modeling*. The implementation aware SDF graph can reflect these hardware constraints with extra back edges, initial tokens, and the worst case execution times (WCET) of the actors.

The actors in the implementation aware SDF graph are mapped onto some hardware units in FLORA. These hardware units are all configured and controlled at run time by MC outside FLORA. If there are multiple jobs mapped on FLORA, MC is also in charge of scheduling them. To create a scheduler instance in MC for every single hardware unit of FLORA, we have to face several hurdles, such as limited communication bandwidth between MC and FLORA, the limited processing power of MC and the awful amount of scheduling workload. In order to overcome these constraints, we try to avoid treating each single hardware unit as the basic scheduling unit and try to get rid of low level scheduling (schedule the tasks working at small granularity). To achieve this, a combined Scheduling View (SV) is derived from a set of implementation aware SDF graphs that are going to be mapped on FLORA. This SV is nothing more than a hardware partition of FLORA. Each partition inside SV consists of several hardware units that can be treated as a single scheduling unit to reduce the scheduling overhead. The combined SV is shared by all the jobs mapped on FLORA. Once we have a combined SV, we can virtually map the implementation aware graph on the scheduling units of the combined SV. Finally, clustering and coupled scheduling (to be introduced) can be applied to the implementation aware SDF graphs after virtual mapping. Clustering (to be introduced) and coupled scheduling can transform the SDF graph from steaming-level granularity (i.e. a token is a bit or byte) to block-level (i.e. a token is thousands bit or byte) granularity. This enables us to analyze the SDF graph in a much higher level without losing any real-time related information. It also helps the online scheduler to schedule the tasks without seeing any detailed operations. Till this step, the offline scheduling is finished.

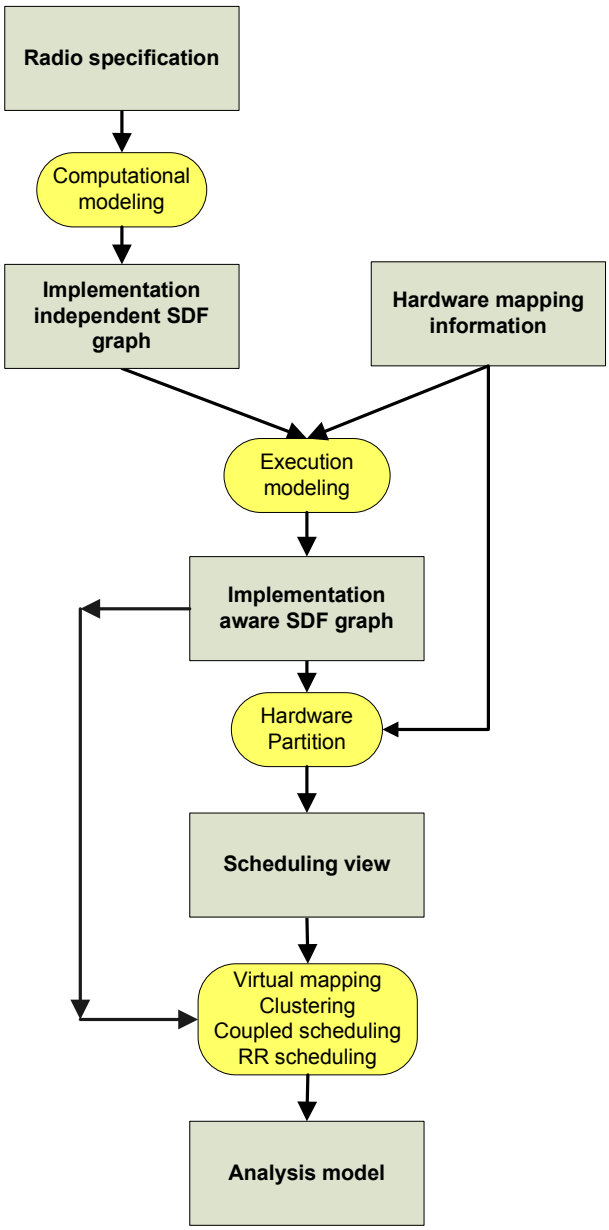

Figure 3-1 Compile time design flow

During run time (Figure 3-2), the real-time constraints force the scheduler to make the scheduling decision in a short time for various combinations of the jobs. Round Robin scheduler, which is non-preemptive, is chosen for every scheduling unit to handle inter-job scheduling during run time. Besides Round Robin scheduler, an overall resource manager is also employed to do admission control to ensure the resource provision and real time performance of each job.

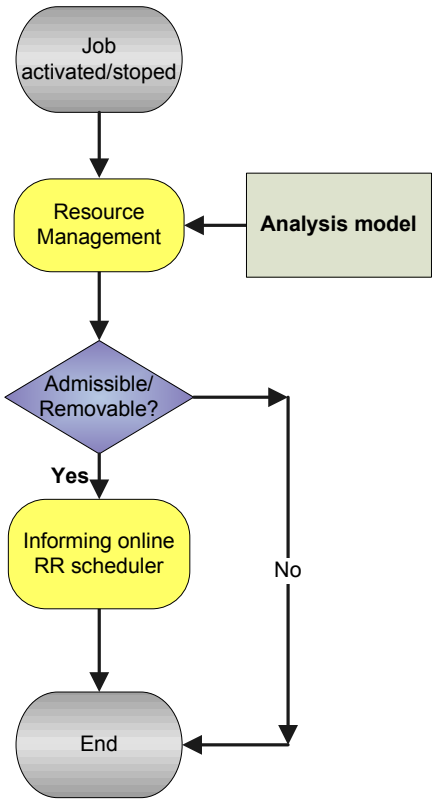

Figure 3-2 Run-time scheduling and resource management

After the Introduction, in section 3.2, the techniques for execution modeling are described. Next, the method to derive the scheduling view is proposed. A modeling technique called clustering is described in section 3.4. In section 3.5, we introduce several scheduling models and describe their properties. We will also highlight the benefits of using them. The last section focuses on resource manager. The resource model and admission rules will be explained in detail.

3.2 Execution Modeling

The radios are well described by the standardized specifications. For each radio, a so-called implementation independent SDF graph can be created from its specification. This graph only specifies the functional behavior of the radio and doesn't take hardware platform that it will be mapped on into account. There are no buffer constraints or processing times. The FIFO channel between two actors is infinite. Figure 3-3 is the implementation independent SDF graph for the channel decoding stage of DVB-SH. DVB-SH is a physical layer standard for delivering IP based media content and data to handheld terminals such as mobile phones or PDAs. It can work in several modes. The example we are using is in 8k, 16QAM mode. The code rate (CR) of the *De-Puncturer* ranges from 3 to 10.

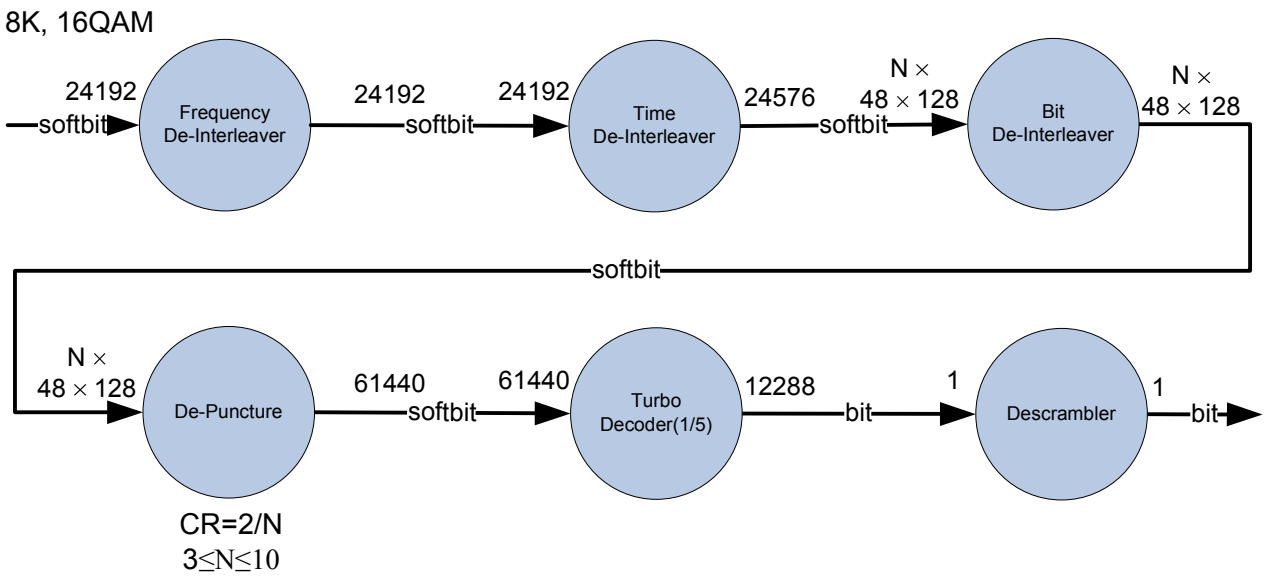

Figure 3-3 Implementation independent SDF graph for DVB-SH

In the implementation stage, the radio will be mapped onto the hardware platform. The implementation aware graph of the radio is created based on the implementation independent graph combined with the hardware mapping information. Some actors will be split or merged due to the hardware constraints and the execution times of the actors will be annotated. Importantly, the FIFO channel between two actors is not infinite any more, since the size of memory or buffer that the channel is mapped on must be limited. The back edges with the initial tokens on them are often used to model the buffer size constraint. The number of the initial tokens on the back edge represents the size of the buffer between two actors. We depict an example of this in Figure 3-4. If there were no back edge from B to A, then A could fire independently of the consumption times of B (i.e. A doesn't need B to release buffer space). Suppose the buffer size between A and B is one token. In Figure 3-4, there is a back edge with 1 initial token between them to simulate the buffer constraint. The first execution of A consumes the initial token and the input token. After the first execution, A can't fire again even there are input tokens available. The back edge with an initial on it forces A to keep waiting until B finishes its execution and stores 1 token on back edge. This behavior is just as same as when the buffer size between A and B is 1.

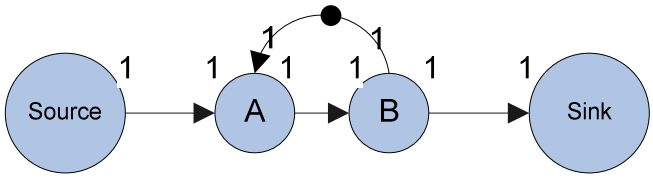

Figure 3-4 Back edge example

Compared with the implementation independent SDF graph, the implementation aware SDF graph for DVB-SH in Figure 3-5 is more restricted. We set the CR of *De-Puncture* as 2/9. Only 1 modes of DVB-SH is modeled out of several modes. The corresponding names and execution times of the actors are shown in Table 3-1. The rectangles behind the actors indicate the hardware units that the actors are mapped onto. Some actors are added to model the behavior of the job after mapping. For instance, the *De-Interleaver* can only perform either reading or writing at a time. Therefore, the actor *Time De-Interleaver* is split to 2 actors: input actor A2 and output actor A3. For *Turbo decoder* in FLORA, it first reads a block of tokens and then starts processing them. To model this instant reading behavior, A6 is added with a zero execution time. Back edges with initial tokens are used to represent the limited buffer size. The tasks are also annotated with execution times that are decided by processing power of hardware.

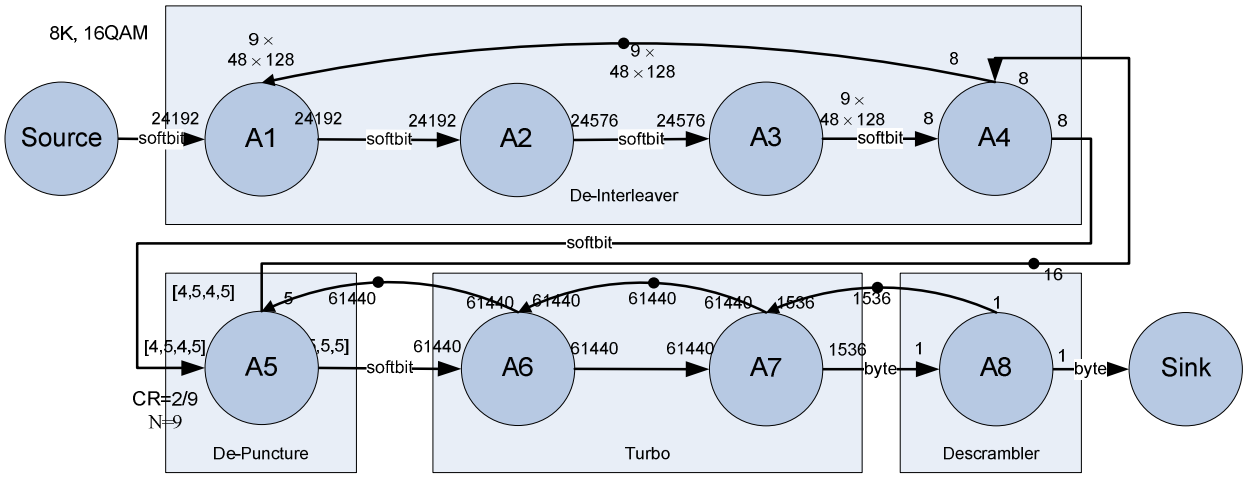

Figure 3-5 Implementation aware SDF graph of DVB-SH

Actor	Corresponding Name	Execution Time
A1	Freq. De-Interleaver. In	62us
A2	Time De-Interleaver. In	26us
A3	Time De-Interleaver Out	26us
A4	Bit De-Interleaver	20ns
A5	De-Puncture	5ns
A6	Turbo In	0
A7	Turbo Out	108us
A8	De-Scrambler	50ns

Table 3-1Actors and their corresponding names

The implementation independent SDF graphs and implementation aware SDF graphs for DVB-T and LTE are depicted in the appendix.

3.3 Scheduling View

As we mentioned in section 2.1, how to reduce the scheduling overhead is a challenge. In this section, we focus on how to derive an hardware partition from a set of SDF graphs, such that several hardware units inside a partition can be treated as a single scheduling unit and the hard-real-time performance is still guaranteed. Before we go into detail, we first introduce several definitions.

A ***Hardware Unit (HU)*** is an atomic (with respect to actor mapping) hardware block in the hardware platform. The HUs in FLORA are *De-Interleaver, De-Puncture, Viterbi, ReedSolomo, Turbo, De-Scramble,* and *CRC.*

We classify the HUs according to the types of their input/output buffers. *De-Interleaver* has a big input buffer but no output buffer. It is the only HU that can read the data from external memory outside of FLORA. We refer to it as an **input-type** HU. Some HUs such as *De-Puncturer, Viterbi, ReedSolomon, De-Scrambler, and CRC,* have a small input buffer and output buffer, we refer to them as **streaming-type** HUs. The HUs like *Turbo* have both a big input and output buffers. We refer to them as **block-type** HUs. Due to the expensive cost of on-chip memory, the input/output buffer size of each HU in FLORA just meets or slightly exceeds minimum required buffer size. "The minimum required buffer size" is defined as the smallest buffer that can store the number of tokens which enable an atomic firing of actors for the relevant applications. Therefore, the HU type also reflects the granularity of the actors mapped to it.

Original Mapping (OM) is a function that takes an actor of the implementation aware SDF graph as the input and produces a HU that the actor is executed by. For instance, In Figure 3-5, $OM(A1) = De-Interleaver$.

A ***Functional Unit (FU)*** is a list of connected actors. The actors in a FU are sorted in the same order as the order in which the actors are executed. The FU is expressed by a pair of parentheses with some actors in it. For instance, $(A4;A5)$ is a valid FU because they are sorted in the correct order. But $(A5;A4)$ is invalid because the order is reversed.

An ***Iteration (I)*** is a list of connected FUs. It starts from the actor that reads the data from the external memory, and ends at the first actor that stores the data back to the external memory. The external memory is big enough for buffering. Therefore, an iteration out of a job can be isolated and scheduled individually. The content of the iteration will only be decided once we have the knowledge of hardware platforms. A job can has several iterations. The Iterations can be derived from the implementation aware graph. It is expressed as a pair of square brackets with FUs in it.

A ***Job Structure (JS)*** is a list of iterations.

FUs, iterations and JS of a job can be derived from the implementation aware SDF graph. The method to derive them is called *chaining*.

Chaining (C) is a function that takes the implementation SDF graph G as the input and produces a JS. The function is working as follows:

```
a_1 ··· a_n are the actors in the implementation SDF graph G and are sorted in the same
order as the order in which they are executed.
m=1;
list<actor> fu;
list<FU> I;
list<Iteration> js;
fu.append( a_1 );
for i=2..n {
     If type(OM(a_i)) ==block-type{
         I.append(fu);
         fu.clean_all();//Find an FU, start a new one
         fu.append(( a_i ));
     }
     If type(OM(a_i)) ==steaming-type{
         if( type(OM(a_{i-1})) ==block-type){
                      I.append(fu);
                      fu.clean_all();//Find an FU, start a new one
                      fu.append( a_i );
         }
         else
                      fu.append( a_i );//update current FU
     }
     If type(OM(a_i)) ==input-type
         if( type(OM(a_{i-1})) ==input-type)
                      fu.append( a_i );//update current FU
         else{
                      js.append(I);//Store the iteration in JS
                      I.clean_all();//Find an Iteraion, start a new one
                      fu.clean_all();
                      fu.append( a_i )//Find an FU, start a new one
            }
     }
}
I.append(fu);
js.append(I);
```

If we apply the chaining to the DVB-SH example, we will get following results:

FUs: $(A1; A2; A3; A4; A5), (A6), (A7) and (A8)$

Iteration: $[(A1; A2; A3; A4; A5); (A6); (A7); (A8)]$

JS: $\{[(A1; A2; A3; A4; A5); (A6); (A7); (A8)]\}$

A ***Hardware Segment (HS)*** is a set of HUs. These HUs can be connected by a configurable matrix network. A FU out of a job can be virtually mapped onto a HS. For instance, $(A4;A5)$ is a FU of the example in Figure 3-5. The corresponding HS is $(De-Interleaver, De-Puncture)$, on which $(A4;A5)$ can be virtually mapped.

A ***Scheduling View (SV)*** is a set of HSs that are mutual exclusive to each other. SV is a partition of all the hardware units. A HS of the SV, consisting of several HUs, is also called a **scheduling unit**. In our approach, an online scheduler is assigned to each scheduling unit to handle scheduling of the tasks mapped on it. The scheduler in the MC will treat the scheduling unit as a basic resource unit. If any HU inside a HS is busy, we say this scheduling unit is busy. The HUs of a HS can run in parallel implicitly to achieve streaming-level pipeline parallelism. The HSs can also run in parallel to achieve better streaming-level or block-level parallelism.

SV impacts the scheduling ability of the system in the following aspects:

- Intra-iteration pipelining. If a SV allows each FU in an iteration to be virtually mapped on its own HS without sharing this HS with other FUs, then the FUs in this iteration can avoid the conflicts on scheduling unit. They can run in parallel and be scheduled by different online schedulers. A maximal intra-iteration pipeline capability is achieved.
- Inter-iteration parallelism. If 2 or more FUs from different iterations don't have resource conflicts, in principle, there exists an SV which allows these FUs to be mapped on different HSs. Then, these FUs can run in parallel, and a maximal inter-iteration parallelism is achieved.
- The number of schedulers. An online scheduler will be assigned to each HS or scheduling unit. Therefore, Fewer HSs in an SV means fewer online schedulers and furthermore, the scheduling overhead is also reduced.

Different SVs may have different impacts on system scheduling ability. On one extreme side, if the SV only consists of one HS which contains all the HUs, there will also be only 1 scheduling unit and the scheduling overhead is minimized, but in the meanwhile, the system loses its pipeline and parallelism capability. For example, if we treat all the HUs inside FLORA as a single scheduling unit, the virtual mapping of iteration $[(A1;A2;A3;A4;A5);(A6);(A7);(A8)]$ in DVB-SH will be the one shown in Figure 3-6.

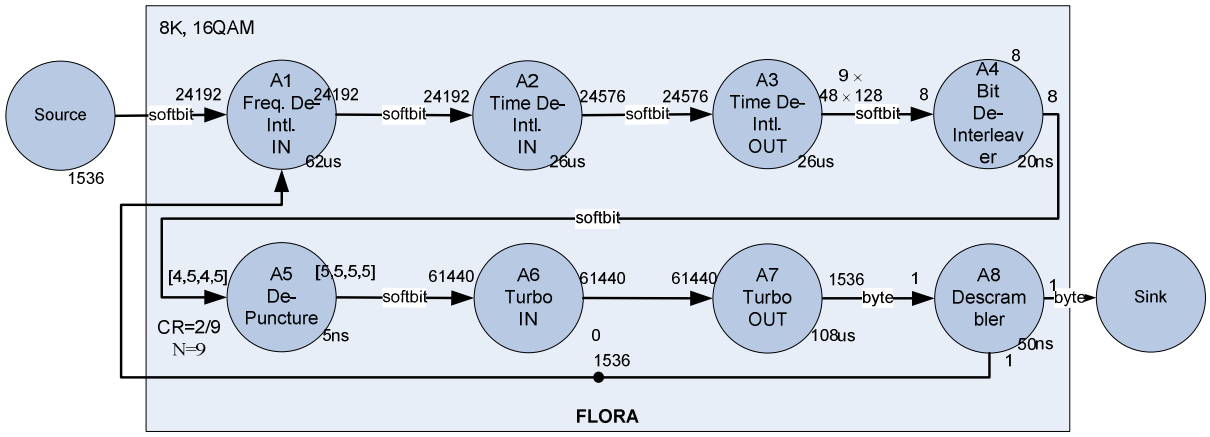

Figure 3-6 Single scheduling unit example

There will be only one run-time scheduler for FLORA. It only starts the new execution once the old one is done. The execution of the DVB-SH is in this way: First, the source produces 24192 tokens, and then the scheduler starts the execution of A1. A1, A2 and A3 are sequentially executed without any pipeline parallelism. After A3 produces a block of tokens, A4 and A5 (They are executed by different hardware as shown in Figure 3-5) start streaming the tokens to the input FIFO of Turbo simultaneously. Once all the input tokens are available, A6 takes all the input tokens from the input FIFO, and pass it to A7 for processing. After a while, A7 produces all the tokens and A8 steams the tokens to sink. When A8 finish streaming all the tokens, the source can produce the tokens for the next round execution and the scheduler starts the execution of the iteration again. It is noticeable that A1 can't start next round streaming until A8 consumes 1536 tokens of the previous round. The reason is that the scheduler treats FLORA as a scheduling unit or single HS. Before A8 finishes streaming 1536 tokens, from the view of the scheduler, the FLORA is still busy, thus, it can't start a new execution.

On the other extreme side, if each HS in SV consists of only one HU, there is more flexibility for the intra-iteration pipeline and inter-iteration parallelism, but we have to assign a run-time scheduler to every single HU. As a consequence, the scheduling overhead will increase. For example, if every hardware unit is a scheduling unit in the SV, the virtual mapping will be different from the example in Figure 3-6, but same as the example in Figure 3-5. For each scheduling unit, there is an online scheduler. The execution of the iteration DVB-SH is in this order: A1, A2 and A3 are executed sequentially. Then A4 and A5 starts streaming the tokens in pipeline. A4 and A5 stop streaming when the 61440-tokens input buffer of A6 is full. At this time, A1 starts execution for next round and in the meanwhile A6 starts execution. A6 passes a block of tokens to A7. Next, A7 processes all the tokens. Finally, A8 streams tokens produced by A7. In this case, there is a block-level pipeline during the execution. The throughput of system is higher than the example in Figure 3-6. However, 4 schedulers, instead of 1, are deployed and the scheduling overhead is increased.

In order to find a proper SV, we need to make a trade off among inter-iteration parallelism, intra-iteration parallelism and scheduling overhead. In our approach, we are looking for the SV with fewest hardware segments that offers maximal intra-iteration pipeline and inter-iteration parallelism.

Based on these criteria, we select the SV according to following three rules. For a given set of iterations,

1) Non-resource-conflicting FUs can only be mapped on different HSs of the SV.

2) An FU can be mapped to several HSs. If a FU is mapped to different HSs, it can only be executed when these segments are all free. We call this coupling and it requires synchronization between scheduling units.

3) Select solution with fewest HSs in the SV.

1st rule guarantees maximal intra-iteration pipelining and inter-iteration parallelism. 2nd rule indicates that if a FU is mapped to different HSs, couplings are needed among these HSs (scheduling units). 3rd rule guarantees the least amount of scheduling overhead, based above 2 rules.

The SV solution for a single iteration can be derived from a direct virtual mapping of all the FUs. We refer to this SV solution as single-iteration SV. For instance, the iteration $[(A1;A2;A3;A4;A5);(A6);(A7);(A8)]$ in Figure 3-5 can be directly mapped on the SV$[(De-Interleaver, De-Puncturer),(Turbo),(De-Scrambler)]$. This SV offers best intra-iteration pipeline capability with fewest HSs or scheduling units. Under this mapping, the system scheduling ability is as same as the one in Figure 3-5, but with only 3 scheduling units instead of 4. The scheduling overhead is reduced.

For multiple iterations, the combined SV solution can be derived from several single-iteration SVs using certain algorithm. Our algorithm is shown below. It has 2 steps: break down and merge, as shown in Figure 3-7. The break down step is used to guarantee maximal intra iteration parallelism. For instance, assume the SV for iteration I_1 is SV1 $[(De-Interleaver, De-Scrambler)]$. I_1 requires $De-Interleaver$ and $De-Scrambler$ to work simultaneously at streaming level. The SV for iteration I_2 is SV2 $[(De-Interleaver),(Turbo),(De-Scrambler)]$. I_2 requires $De-Interleaver$ and $De-Scrambler$ to work independently and in the mode of block pipleline. If the combined SV is $[(De-Interleaver, De-Scrambler),(Turbo)]$, then for I_2, $De-Scrambler$ and $De-Scrambler$ are in the same scheduling unit, therefore they can't be scheduled independently and they lose the block-level pipeline capability. Thus, according to requirement of SV2, the we first break down the SV1 $[(De-Interleaver, De-Scrambler)]$ to be SV1' $[(De-Interleaver),(De-Scrambler)]$. We also break down SV2 according to the requirement SV1. SV2' is same as SV2. Finally, SV1' and SV2' can be easily merged into a combined SV.

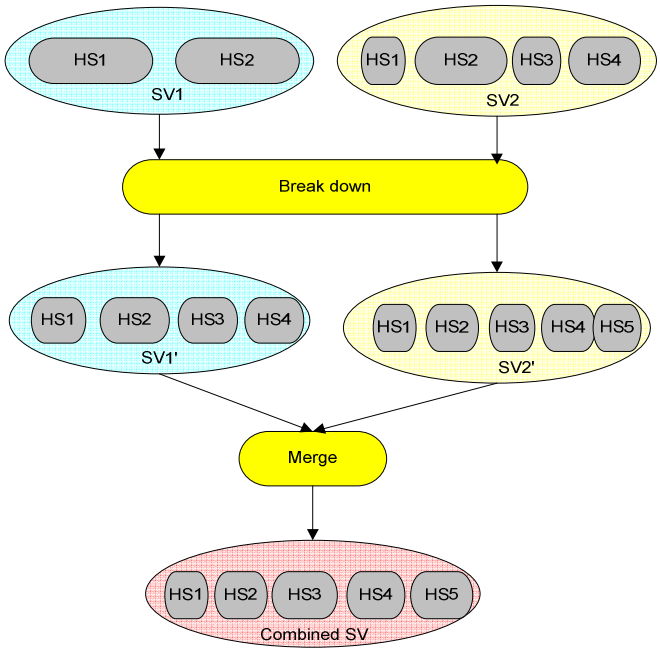

Figure 3-7 Break down and merge

The pseudo code of the algorithm is shown below.

```
Given single-iteration scheduling views sv1 = [hs1^1, ··· hs_m^1], sv2 = [hs1^2, ··· hs_n^2].
Step1: Break down sv1 and sv2 by function break:
sv1' =break(sv1,sv2);
sv2' =break(sv2,sv1);
Step2: Merge two sv into a combined SV csv:
csv=merge( sv1', sv2' ).
function break(sv1,sv2){
        SV s=sv1;
        HS tem = ∅ ;
        for i=1..lengthof(sv1){
                tem = ∅ ;
                for j= lengthof (sv2)..1{
                        if (j==1){
                                s.delete( hs_i^1 );
                                s.add( hs_i^1 \ tem );
                                tem = ∅ ;
                        }
                        else{
                                s.add( hs_i^1 ∩ hs_j^2 );
                                tem= tem ∪ (hs_i^1 ∩ hs_j^2) ;
                        }
                }
        }
        return s;
}
```

```
function merge(sv1,sv2){
            int counter;
            SV s=sv1;
            for i=1.. lengthof (sv2){
                        counter=0;
                        for j=1.. lengthof (sv1){
                                    if( hs_i^1 ∩ hs_j^2 ≠ ∅ ){
                                                s.delete( hs_i^1 );
                                                s.add( hs_i^1 ∪ hs_j^2 )
                                                counter=1;
                                                break;
                                    }
                        }
            }
            if(counter==0)
                        s.add( hs_j^2 );
      }
      return s;
}
```

The single-iteration SVs and combined SV for DVB-SH, DVB-T and LTE is derived as shown in the appendix A.3. .

3.4 Clustering

In section 3.3, we introduce the derivation of SV from a set of the implementation aware SDF graphs. Once we have the SV, we can virtually map the tasks to HSs or scheduling units. If the actors of a FU are mapped to the same HS or scheduling unit, streaming pipeline will be implicit among these actors. In order to reduce the scheduling overhead and get rid of unnecessary detailed information to simplify the analysis, we try to transform fine-granularity SDF graph into coarse-granularity SDF graph. To achieve this, a modeling technique called clustering is employed.

To apply clustering to the actors in fine-granularity SDF graph, some conditions need to be met.

- The actors are working at same granularity (i.e. bit, byte).
- The actors are originally mapped on different hardware units and virtually mapped on the same scheduling unit.
- The buffers before and after these actors are big enough (in block level, i.e. thousands bits, bytes).
- The buffers' sizes between actors are at least 2 tokens.

Clustering will merge these actors into a single task. The WCET of the new task will be calculated according the Latency Rate Severs (LRS) mentioned in [14] as follows:

$A1 \cdots An$ are the actors going be clustered. The WCETs of them are $T_{A1}, \cdots, T_{An}$ respectively. k is the number of the firings to finish streaming a block of input tokens. The WCET of new task after clustering is $T_{A1} + \cdots + T_{An-1} + k \times Max(T_{A1,} \cdots, T_{An})$. For example, suppose there is an iteration $[(A1; A2); (A3)]$ and the scheduling view is $[(De-Interleaver, De-Puncturer), (Turbo)]$. A1 and A2 are originally mapped on *De-Interleaver* and *De-Puncturer* respectively, and now they are virtually mapped on scheduling unit $(De-Interleaver, De-Puncturer)$. $A3$ is originally mapped on *Turbo* and now virtually mapped on scheduling unit $(Turbo)$. The implementation independent graph and graph after virtual mapping are shown in Figure 3-8. The back edge with 2 initial tokens on it indicates the buffer size between A1 and A2 is 2. The back edge with 1000 initial tokens on it indicates the buffer size between A2 and A3 is 1000.

We want to emphasize that even A1 and A2 are virtually mapped on same scheduling unit, but they are running in parallel implicitly, since they are executed by different hardware units. The hardware units inside a scheduling unit have streaming pipeline capability.

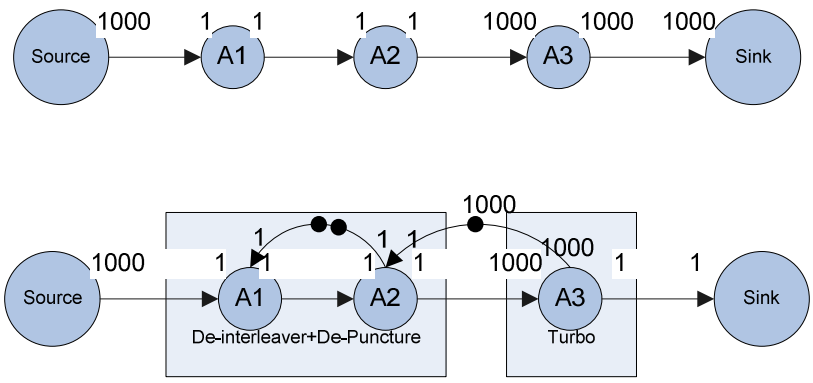

Figure 3-8 Graphs before clustering

If we want to cluster A1 and A2 as a single task working at block level, we first have to answer a question: what's the WCET of the task after clustering? The WCET of A1 and A2 are T_{A1} and T_{A2} respectively. The buffer size between them is 2. The task after A2 is A3. There is a 1000 tokens sized buffer between them. The behavior of A3 implies that it can only start execution after A1 and A2 complete streaming 1000 tokens to its input buffer.

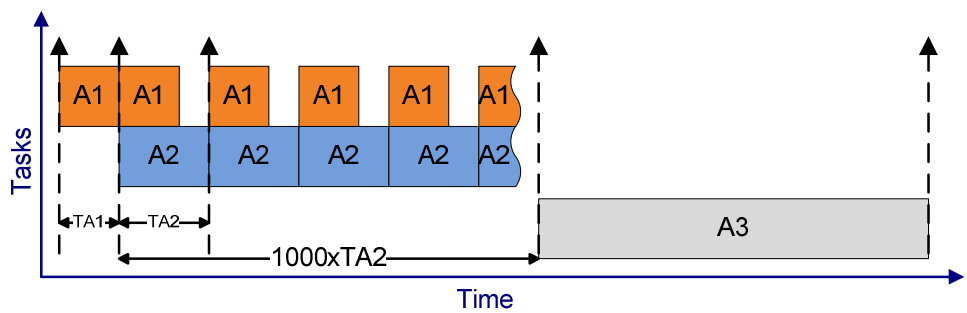

Figure 3-9 Time diagram of the execution

The execution of above example is shown in Figure 3-9. If we cluster this 1000 times repeated streaming behavior of A1 and A2, the buffer pressure before and after A1 and A2 won't change. The SDF model will be still conservative. After the clustering, the result is shown in Figure 3-10. Given that A1 and A2 can be executed in pipeline, the WCET of new task Ac is $T_{A1} + T_{A2} + 1000 \times Max(T_{A1}, T_{A2})$.

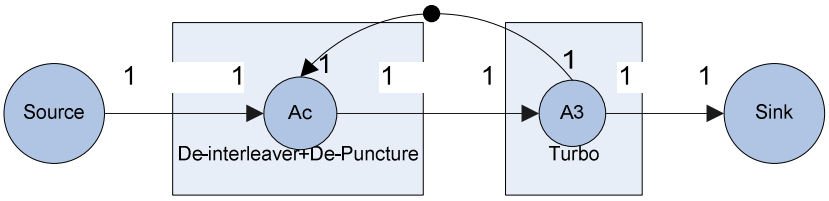

Figure 3-10 SDF graph after clustering

Compared with Figure 3-8, the SDF graph after clustering in Figure 3-10 omits streaming level detail and focuses on block level pipeline. The clustering will make scheduling analysis much easier. We will explain this in the next section.

Clustering can transform the fine-granularity SDF graph to coarse-granularity SDF graph. But it only works under strict conditions. For some complex SDF graphs (i.e. non-linear graph, cyclo-static graph), clustering can't estimate the execution times of the actors after transformation accurately. In this master graduation project, the execution times of some actors after transformation, are based on the real experimental data.

3.5 Scheduling Models

On a multiprocessor platform, scheduling the jobs often involves several steps. First, the tasks of a job need to be assigned to one or more processors. Then the execution order of the tasks has to be made. Next, the start time of every task needs to be decided by the scheduler. These scheduling decisions can be made either during compile time or during run time. In [10], several scheduling strategies have been discussed. The fully dynamic scheduler makes all the scheduling decisions at run time. It can handle highly dynamic program behavior by changing the order in which tasks run, and by adjusting processing loads during run-time. However, the cost of such run-time scheduling decisions is very high. The fully static scheduling has least amount of run-time overhead, but all the scheduling decisions have to be made in the compile time. If there are too many jobs and they can be mixed dynamically at run time, it is a challenge to guarantee that all the tasks will meet their hard-real-time deadlines.

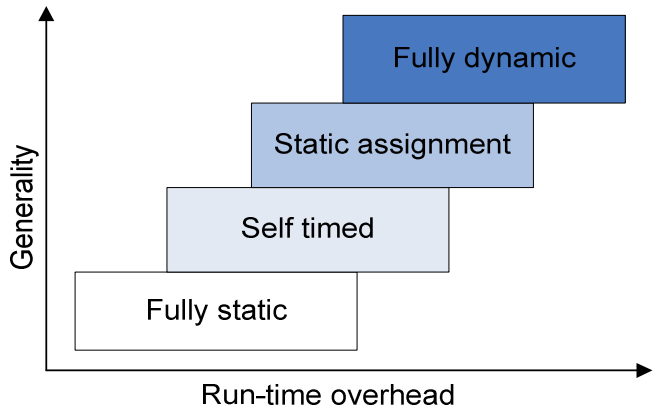

Figure 3-11 Trade off between generality and run-time overhead

FLORA is our target hardware platform. It is not a fully programmable multiprocessor platform, but a configurable hardware accelerator, consisting of several configurable hardware units controlled by an external MC via the control bus. Due to the non trivial run-time overhead, we can't fully dynamically schedule all the tasks during run time. The fully static scheduling is not a choice either, since there are many kinds of jobs combinations during run time. If we measure the scheduling methods by run-time overhead and generality, there are some other scheduling methods between fully dynamic and static scheduling as shown in Figure 3-11. In order to balance generality and run-time overhead, Round Robin scheduling and coupled scheduling are used.

3.5.1 Round Robin Scheduling

In the run-time, we need a run-time scheduler to decide the execution order and the start time of the tasks virtually mapped on a scheduling unit. Round Robin (RR) scheduler is chosen to be such the run-time scheduler, because it is non-preemptive, easy to implement and starvation-free.

RR scheduling can be non-preemptive. FLORA doesn't support preemptive scheduling and RR scheduling is a good candidate.

RR scheduling is not a complex mechanism and easy to implement. In our approach, the online scheduler should be as easy as possible, such that the run time decision can be made in a short time.

RR scheduling is starvation-free. It constantly checks a list of tasks. If the input data is available on the input channel of a task, the RR scheduler will start its firing. If input data is not ready, the RR scheduling will skip it and check the next task in the list. This property indicates that we will always have the upper bound of waiting time under RR scheduling. This upper bound of the waiting time is called the worst case waiting time (WCWT). The WCWT of a task *a* mapped on the scheduling unit SU is equal to the sum of WCETs of the other tasks in the RR list. The equation is shown below.

$$WCWT(a) = \sum_{Map(x)=SU, x \neq a} k_x \times WCET(x) \qquad 3.1$$

Where $WCWT(a)$ is the WCWT of *a*, k_x is the number of times an task x appears in the RR list, and $WCET(x)$ is the WCET of task x .Based on WCWT, the worst case response time (WCRT) of actor a can also be derived as shown in below.

$$WCRT(a) = WCET(a) + WCWT(a) \qquad 3.2$$

Where $WCRT(a)$ is the WCRT of task *a*, $WCET(a)$ is the WCET of task *a* and $WCWT(a)$ is the WCWT of *a*.

As shown in Figure 3-12, in the SDF graph, the RR scheduling can be incorporated with an actor W to model the waiting time of task to get the processing resource of scheduling unit SU (The square box in Figure 3-12). Another actor *a* is modeling the real processing time of task. This modeling technique is important for the resource awareness in run time.

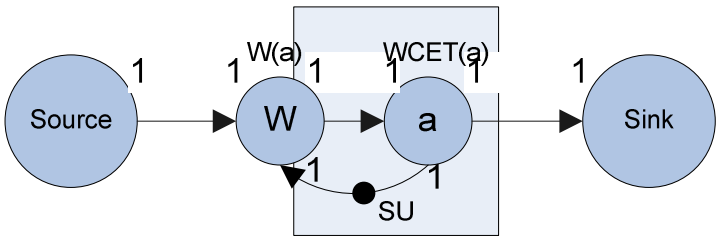

Figure 3-12 Round Robin SDF example

3.5.2 Coupled Scheduling

Sometimes the actors out of the same FU, which are pipelined and working at small granularity, are virtually mapped on different scheduling units. As we mentioned, every scheduling unit is under the control of its own run-time scheduler. Due to this, the actors are decoupled and scheduled by more than one scheduler. The streaming flow between them is broken. However, we want to schedule the actors in same FU at coarse granularity (i.e., using the clustering technique) to reduce the scheduling overhead. Coupled scheduling is used to solve this contradiction. Coupled scheduling forces the actors, which are out of same functional units and mapped on different scheduling units, to synchronize their first firings, and stream rest of the data in a pipeline mode.

An example is given in Figure 3-13. Suppose the iteration is $[(A1; A2)]$ and the scheduling view is $[(De-Interleaver),(De-Puncturer)]$. In the implementation independent SDF graph, A1 and A2 are streaming data at small granularity (i.e. bit, byte). They are virtually mapped on $(De-Interleaver)$ and $(De-Puncturer)$ respectively. The buffer size between A1 and A2 is 2 tokens. A1 and A2 are scheduled by different schedulers. We are trying to avoid the low level scheduling, such that the scheduler doesn't have to decide the start time of each firing. The clustering serves this purpose. It omits the detailed information, and binds a block of firings together. Therefore, the scheduler only needs to worry about the starting time of the first firing. However, clustering can't be directly applied to the actors virtually mapped on different scheduling units because some of the conditions mentioned in section 3.4 are not met.

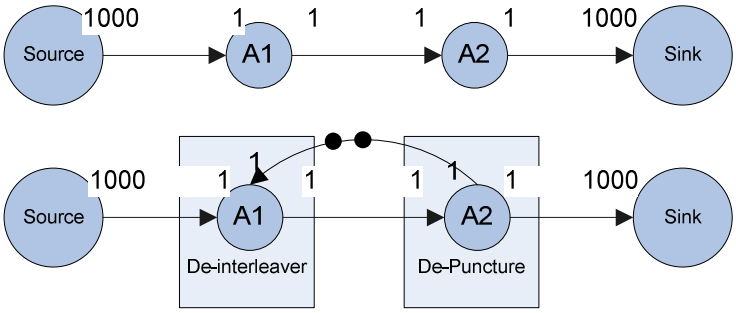

Figure 3-13 SDF graphs before coupled scheduling

If we use the coupled scheduling to synchronize the first executions of A1 and A2, then the online schedulers only need to decide the start time of first firings of A1 and A2 and make the next decision after the completion of streaming 1000 tokens. The coupled scheduling is shown in Figure 3-14. After the source produces input tokens, A1 starts waiting until the online scheduler gives the processing resource (There might be other actors mapped and running on $(De-Interleaver)$ and $(De-Puncture)$). The actor that models the waiting times of A1 is named W1. When W1 is done, A1 gets the processing resource, and in the meanwhile, activates the actor that models waiting times of A2, named W2. The edge from W2 to A1 indicates that A1 won't start until W2 is done (A2 gets the processing resource). W2 continues until the online scheduler gives the processor to A2. Now, A1 and A2 are holding their processing resources and start their first firing. The first firings are synchronized.

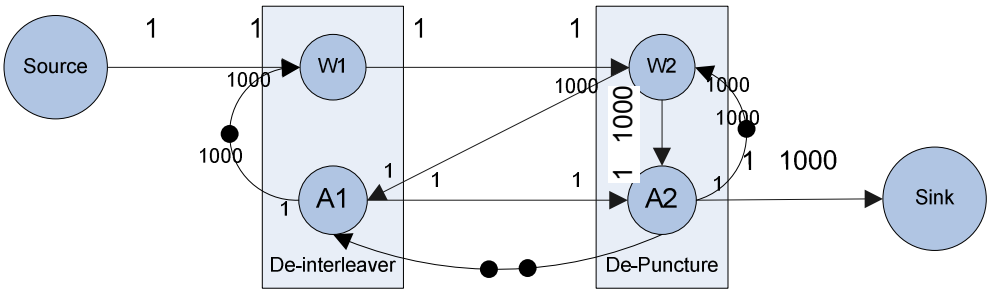

Figure 3-14 SDF graph after coupled scheduling

After the coupled scheduling, we can apply clustering on A1 and A2. The result is depicted in Figure 3-15. Note, the graph in Figure 3-15 is self-timed executed. Therefore, A1'and A2' will start simultaneously.

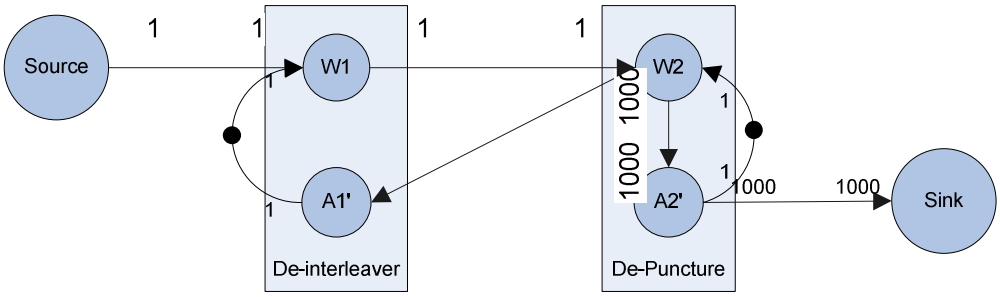

Figure 3-15 SDF graph after clustering

According to the rules of clustering, the WCET of A1' and A2' are $T_{A1'}$ and $T_{A2'}$, where $T_{A1'} = T_{A2'} = 1000 \times Max(T_{A1}, T_{A2}) + T_{A1}$.

In order to keep the jobs being scheduled independent from each other, we want emphasize that the coupled scheduling is only used as an intra-job scheduling. Therefore, the actors need to be coupled are always in the same job and won't cause the deadlock.

The drawback of the coupled scheduling is the cost of synchronization. In the above example, if A2 doesn't get the control of the processor, A1 can't start even it is holding the processing resources. There is a period during which A1 is wasting the processing resources without doing anything except waiting for A2. The cost of synchronization must be taken into account.

3.6 Resource Management

During run time, the end users can start a new radio application at any time. However, the hardware resources of FLORA are limited. If there are too many radios running on FLORA, there is no guarantee for temporal behaviors of all the radios. Therefore, an online resource manager (RM) is designed. Whenever a start request arrives, RM will check whether the new job can be accepted or not, according to the admission rule. If the hard-real-time behavior of new application and the other running applications can still be guaranteed, the RM will inform the online scheduler to scheduler the new application. If not, the start request will be denied by RM.

In order to measure the usage of the hardware resource for each application, we need to create a model of the resource consumption for each application. There are two major resources involved in FLORA: hardware units and the buffer used by Turbo decoder. However, in this thesis, a scheduling unit is treated as a basic processing unit. Therefore we will model scheduling units instead of hardware units as the processing resources. A SDF graph J, which has been clustered and coupled scheduled, is mapped on a combined SV S, where $S = [su_1, \cdots su_n]$. The resource consumption of J can be modeled as:

$$rc(J) = [T_{su_1}, \cdots, T_{su_n}, M_J^i, M_J^o]$$

Where T_{su_1} to T_{su_n} are WCETs of the actors mapped on $su_1, \cdots su_n$, M_J^i and M_J^o are the amount of Turbo's input and output buffer space that occupied by J.

The admission rules can be built on the top of this resource consumption model. Suppose the total amount of input and output Turbo buffers are M^i and M^o respectively, the combined SV is $[su_1, \cdots su_n]$. Let $J_1 \cdots J_m$ be a set of clustered and coupled scheduled SDF graphs, with the resource consumption models $rc_{J_1} \cdots rc_{J_m}$. If a new SDF graph J_{m+1} that has the resource consumption model $rc_{J_{m+1}}$ is admissible, it must meet the following admission rules:

1) WCRT of each task must be smaller than its relative deadline (RD). The WCRT can be calculated according to equation 3.1 and 3.2.

2) $M^i \geq \sum_{k=1}^{m+1} M_{J_k}^i$

3) $M^o \geq \sum_{k=1}^{m+1} M_{J_k}^o$

Once the new job is accepted, the RM will inform the online scheduler to add the new job in the RR list.

4 Implementation

4.1 Introduction

In this chapter, we will show how to implement the modeling and scheduling techniques in the simulation system. The RTL implementation of FLORA is available and the accurate verification can be carried out, but to build the real implementations of jobs (DVB-T, DVB-SH, and LTE) and the schedulers around the RTL implementation is not easy. The trade-off has to be made among the cost, simulation speed and accuracy as shown Figure 4-1.

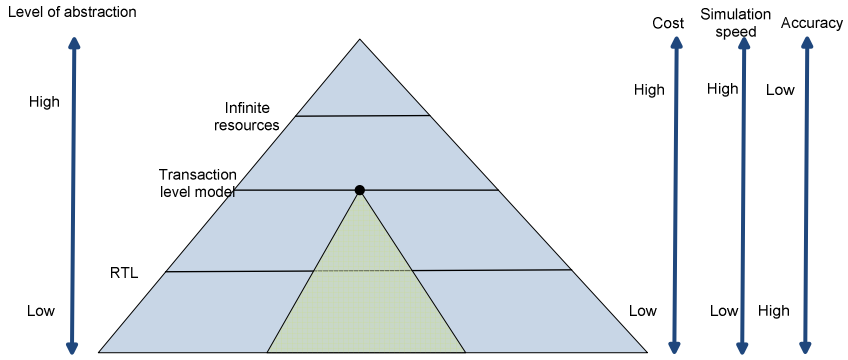

Figure 4-1 abstraction

In order to cut the cost and increase the simulation speed, a simulation system with a moderate accuracy is created by Coware ESL (electronic system level) 2.0 tools [15]. ESL 2.0 refers to a second generation of ESL solutions, which aim to facilitate the design and development of processor-centric, software-intensive products with complex interconnect and memory architectures.

The simulation flow is shown in the Figure 4-2. From the implementation aware SDF graphs, we create the non-functional task models in Coware. The task model we build keeps the same temporal behavior as the behavior of its corresponding implementation aware SDF graph, without specifying any detailed functional operations. The techniques such as coupled scheduling and clustering are applied to these task models. Besides the task models, a FLORA simulation platform is also designed. The task models of DVB-T, DVB-SH, and LTE will be mapped onto this simulation platform. The tasks of run-time RR scheduler and RM are also created and mapped onto simulation platform. The simulation system is formed by all the task models and the simulation platform. It can capture the temporal behaviors of these applications that are mapped onto simulation platform, such that we can verify our approach based on the simulation results.

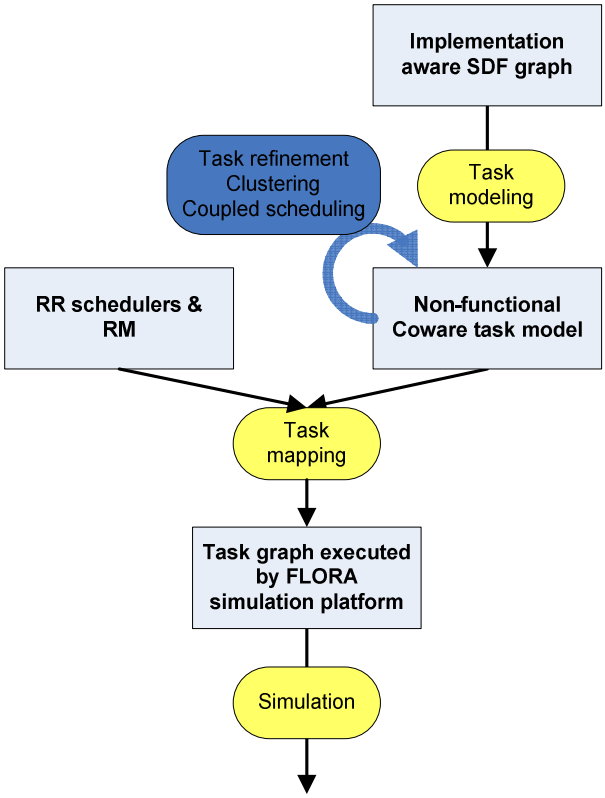

Figure 4-2 Simulation flow

4.2 Simulation Platform

To model FLORA architecture, we use platform creator from Coware ESL 2.0 in GUI or in command line interface (CLI). They are equally functional. But the scalability of CLI is better, since we can use scripts to automate the design. Therefore, our simulation platform is created by platform creator CLI.

Coware ESL provides some standard libraries. To model the FLORA, we will use some components from standard Coware libraries. The components in the Coware libraries include memory, bus, Virtual Processing Unit (VPU) etc. These components have a set of standard interfaces and can be assembled to form a simulation hardware platform. Hardware units of FLORA, such as De-Interleaver, De-Puncture are modeled by VPU. VPU is a processing abstract resource for a number of tasks. It is the model of processing unit to reason about the temporal behavior of the tasks running on it. A VPU has its own task manager. The tasks that run on a VPU are controlled by its task manager. The model of De-Interleaver in the FLORA is created by following codes:

```
::pct::open_library SCML_TM_VPU
::pct::open_library SCML_BL
::pct::open_library GenericIPlib
::pct::open_library PV_BL
::pct::open_library AVF_BL
::pct::instantiate_block SCML_TM_VPU:scml_tm_vpu_no_interrupt . InterLeaver
::pct::set_param_value InterLeaver "Template Arguments" nbr_of_memory_ports 1
```

```
::pct::set_param_value InterLeaver "Template Arguments" BUSWIDTH 64
```

Table 4-1 Open libraries and instantiate blocks

Not only the hardware units, but also the external memory and MC outside FLORA need to be modeled. The external memory can be an instance of memory block in *GeniricIPlib*. The MC can be modeled as another VPU, on which control tasks like RM and the online schedulers are running.

All the components in FLORA, MC, and the external memory, can be connected by bus or FIFO channel. The bus we select is in *AVF_BL* library, which is defined by OCP standard. The OCP bus has a standard interface where the VPUs and memory can be connected. It also has an arbiter to schedule the communication requests from all the VPUs. The bus consumes some cycles to transfer the data from the sender to receiver. In our case, it is from VPU to external memory and vice versa. The VPU can access external memory via the bus, but it must be aware of the range of the addresses where the memory is mapped. The following piece of codes shows how to do the memory map in CLI.

```
#memory map
::pct::add_target    X_MEM/MEM
::pct::add_initiator InterLeaver/p_mem\[0\]
::pct::set_address   InterLeaver/p_mem\[0\]:X_MEM/MEM 0
```

Table 4-2 Memory

After the memory mapping, from the view of *p*_mem port of De-*Interleaver*, the starting address of external memory is 0. The procedure of creating simulation platform is shown in Figure 4-3.

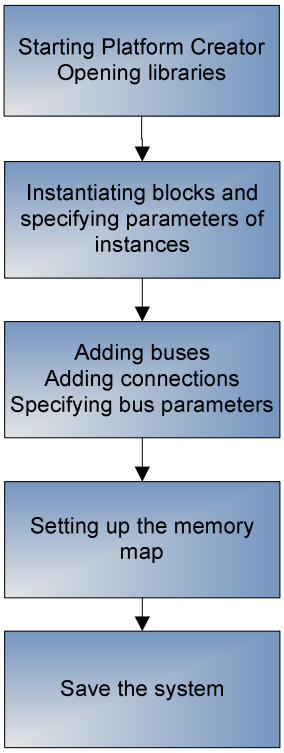

Figure 4-3 Flow of creating virtual platform

There are also some other components such as clock generator and the restart generator. Clock generator decides the frequency of the system. In our case, the FLORA is clocked at 200MHz. The FLORA model in Coware is shown in Figure 4-4. We want to emphasize that the connections between the VPUs in this step are not established. This is because the real FLORA uses a matrix network to connect the hardware units. The actual routing of connections is not determined until the applications are mapped onto the FLORA. Therefore, in the simulation platform, making these connections is left to the mapping step.

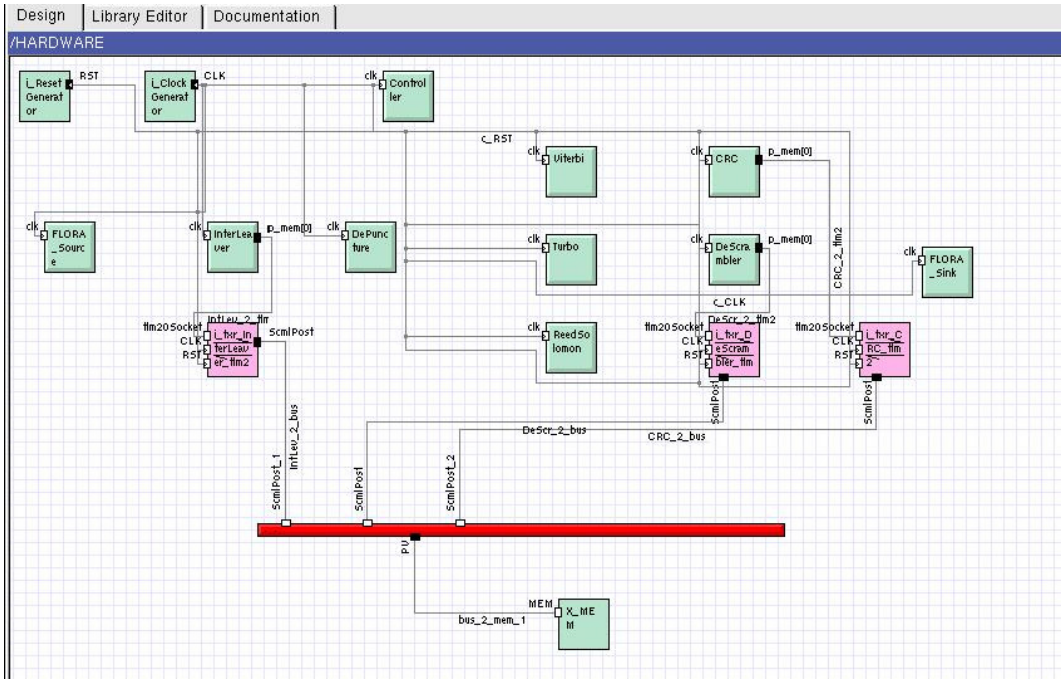

Figure 4-4 FLORA model

4.3 Task Modeling

The term *task modeling* refers to the modeling of a job as a set of tasks that can communicate with other tasks. This model of the job can be either a functional or a nonfunctional model. We are focusing on the scheduling of the tasks, meaning that the functional details are not our main focus. We only need the model that can capture the certain properties of the job, such as the amount of processing time required to execute the task and the amount of data that is required to be transferred to perform certain operations. Task model in Coware is created based on implementation aware SDF graph, where the actors are annotated with execution time and the numbers of tokens consumed and produced are specified. Each actor in SDF graph can be modeled as a single task in Coware. The edges in SDF graph is modeled as the *ste_tm_fifo_channel*. Compared with *scml_tm_fifo_channel* in the standard Coware library, some new APIs are added in *ste_tm_fifo_channel*, such as the function to check the occupied and available space in the FIFO.

The language used to do task modeling is SystemC. A task is modeled as a SystemC thread. For communication with other tasks, this thread is part of a *sc_module*. This module can have ports for communication with other tasks. The communication happens over channels. The software can access a task-modeling API to annotate execution times and traffic to be generated and to pass control over to other tasks. Each task has a priority which can be used by the scheduler and tasks can be grouped in jobs. A job is identified by its job ID. The task-modeling API is the API for communication of the tasks with the task manager. The task manager controls the states of the tasks it manages. There are several states for a task as shown in Figure 4-5.

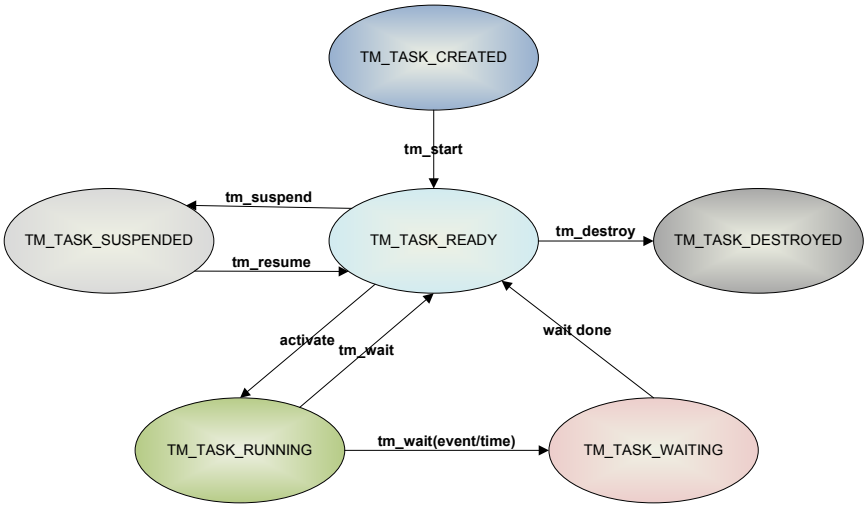

Figure 4-5 Task states

More detailed information about the state of the task can be found in [16]. Once a task graph of an application has been created, it can be mapped unto one or more VPU in the simulation platform. The default mapping of a task graph is that all the tasks are mapped on a single VPU. The sample code of a typical streaming task is shown below.

```
virtual void task(){
    while(true){
         //waiting for the input token and output space
         while(!(p_get->nb_can_get()&&p_put->nb_can_put()))
                       tm_wait(1);
         //consume some time before gets the input token
         tm_consume(delay_before_get);
         p_get->nb_get(data);
         //consume some time for processing the token
         tm_consume(processing_delay);
         //consume some time before puts the output token.
         tm_consume(delay_before_put);
         data_out= process(data);
         p_put->nb_put(data_out);
   }
}
```

Table 4-3 Streaming task model

The task starts with checking whether there is input token in the input channel and output space in output channel. If not, it will stay in the *TM_TASK_WAITING* state for one clock cycle, and then join the ready queue again, waiting for the local scheduler (on the VPU that the task is mapped on) to activate it. It repeats the checking behavior until there are input token and output space. Once the input token is ready and output space is available, the task consumes some time to get the input token, and then start processing it. After the output token is produced, it puts the output token to the output channel. The whole task is an infinite while loop, it repeats streaming single token until there are no input token or no output space. All the tasks are modeled based on the framework in Table 4-3.

For some tasks working at coarse granularity, the amount of input tokens and output space that need to be checked is a block of tokens. After all task models are built in SystemC, we can compile them and an "xml" file will be generated as a task library.

There are still a few steps left: to instantiate the task blocks and connect them by using *ste_tm_fifo_channel.* The tasks are connected in the same order as the order they are executed in. The whole procedure is done in tcl script and platform creator CLI. A sample code of creating DVB-SH application is shown below.

```
::pct::new_project
::pct::open_library SCML_TM_PL
::pct::open_library $::env(STE_LIB)/STE_LIB.xml
::pct::open_library ../Tasks/DVB_SH_Tasks/sh_flora_tasks.xml
::pct::open_library ../Tasks/Scheduler_Tasks/Scheduler.xml
::pct::open_library ../Tasks/RM_Tasks/RM.xml
source ../procedure.tcl

::pct::instantiate_block sh_flora_tasks:sSource /HARDWARE sh_1_Source
::pct::instantiate_block sh_flora_tasks:sSink /HARDWARE sh_1_Sink
::pct::instantiate_block sh_flora_tasks:sSymbolDI /HARDWARE sh_1_SymbolDI
::pct::instantiate_block sh_flora_tasks:sTimeDI_mem /HARDWARE sh_1_TimeDI
::pct::instantiate_block sh_flora_tasks:sBitDI /HARDWARE sh_1_BitDI
::pct::instantiate_block sh_flora_tasks:sDePuncture /HARDWARE sh_1_DePuncture
::pct::instantiate_block sh_flora_tasks:sTurbo /HARDWARE sh_1_Turbo
::pct::instantiate_block sh_flora_tasks:sDeScrambler /HARDWARE sh_1_DeScrambler

intra_connect HARDWARE sh_1_Source p sh_1_SymbolDI p_get f_softbit_8 3024 false
data
intra_connect HARDWARE sh_1_SymbolDI p_put sh_1_TimeDI p_get f_softbit_8 3024
false data
intra_connect HARDWARE sh_1_TimeDI p_put sh_1_BitDI p_get f_softbit_8 3072 false
data
intra_connect HARDWARE sh_1_BitDI p_put sh_1_DePuncture p_get f_softbit_8 2 false
data
intra_connect HARDWARE sh_1_DePuncture p_put sh_1_Turbo p_get f_softbit_8 2 true
data
intra_connect HARDWARE sh_1_Turbo p_put sh_1_DeScrambler p_get f_byte 8 true data
intra_connect HARDWARE sh_1_DeScrambler p_put sh_1_Sink p f_byte 8 true data
::pct::save_system 1_sh_task_graph.xml
```

Table 4-4 Connecting the tasks

The DVB-SH task graph is shown in Figure 4-6. This task graph is derived from the implementation aware graph and they have the same temporal behaviors. The clustering and coupled scheduling have not been imposed on the task graph in Figure 4-6.

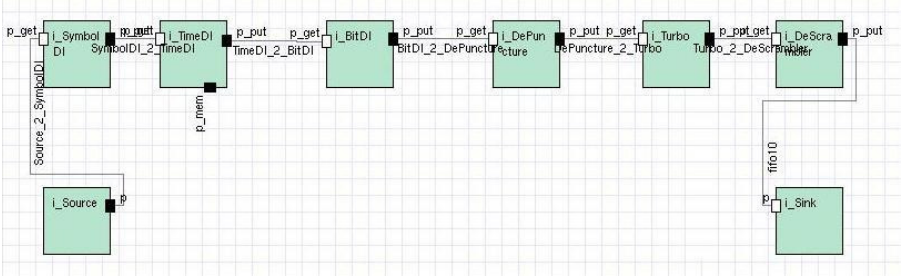

Figure 4-6 DVB-SH task graph

4.4 Mapping

The task graph will be finally mapped onto the simulation platform created in section 4.2. It is a static mapping, since the hardware units are all configurable hardware accelerators which are designed only for specific tasks. Therefore, in the mapping stage, we only need to map the tasks to the corresponding VPUs (hardware units) and make the connections between tasks that are mapped on different VPUs. When communicating tasks are split over multiple VPUs, the original connection has to be refined. Instead of a direct connection, the communication needs to happen over the hardware of the system. For this purpose, a driver is put between the task and the VPU port that connects to the hardware of the system. Driver is a special channel that enables the communication of tasks with the platform and the communication between tasks running on different VPUs. When drivers are connected to the memory ports of the VPU, they need to produce TLM2 transactions. For convenience, the SCML_TM_VPU library provides an scml_tm_post_tlm2_transactor module. This module provides a simple post interface for TLM2 transactions and it handles the TLM2 communication on the VPU ports.

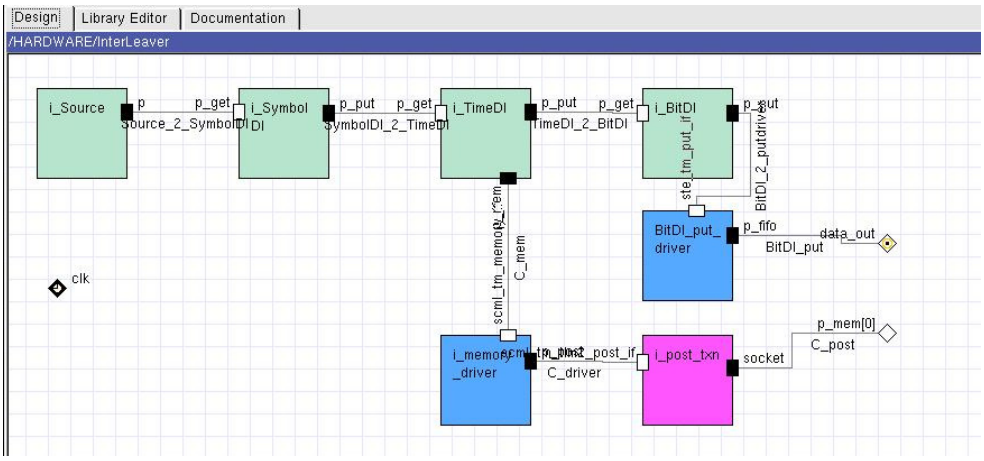

Figure 4-7 DVB-SH De-Inteaver mapping

Suppose we map the task graph of DVB-SH application in Figure 4-6 onto FLROA simulation platform, the tasks such as SymbolDI, TimeDI, and *BitDI* are mapped on *De-Interleaver* VPU block as shown in Figure 4-7. The connection between *BitDI* and *DePuncture* is replaced by the drivers on *BitDI* side and *DePuncture* side. *TimeDI* is communicating with the external memory. A memory driver is assigned. An *scml_tm_post_tlm2_transactor* module named *i_post_txn* is also used to bridge the protocol gap between memory driver and memory port of VPU.

The overview of the mapping is shown in Figure 4-8. Compared with the simulation platform in Figure 4-4, the hardware blocks in Figure 4-8 are connected by *sc_fifo*. The *sc_fifo* together with the drivers in the hardware blocks, act as the communication channel between tasks mapped on different hardware blocks.

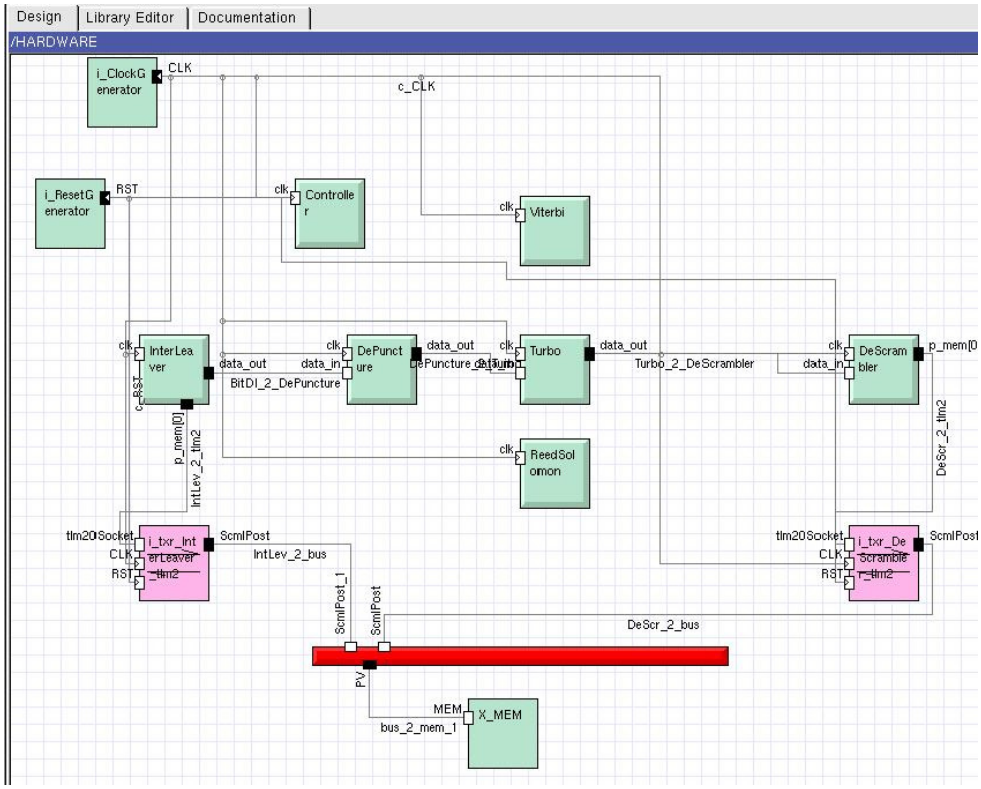

Figure 4-8 overview of simulation system after DVB-SH mapping

4.5 Clustering

Clustering is a technique to transform a streaming-level SDF graph into block-level SDF graph. There are some conditions need to be met before apply the cluster to tasks (mentioned in section 3.4). The task graph model we created in section 4.3 is derived from the implementation aware graph that is working at small granularity. The streaming task model in Table 4-3 is checking if the single input token and space for single output token are available for each execution (firing). If the scheduler has to take care of every checking and firing, the scheduling overhead will be huge. But instead, if we can cluster the thousands of firings of a task, the task only need to check if there are thousands of input-tokens and space for thousands of output-tokens, and then task keeps firing for thousands of times. The scheduler now only needs to decide the start time of the first firing instead of every firing. Therefore, the scheduling overhead reduced is reduced by clustering. If we take streaming task model in Table 4-3 as an example, the clustered model of it is shown in Table 4-5.

```
virtual void task(){
   while(true){
       if(counter == 0){
          while(!(p_get->nb_checkread(data_size_in_softbits)&&p_put->nb_
          checkwrite (data_size_out_softbits)))
          {
            if(!p_get-> nb_checkread(data_size_in_softbits))
                      tm_wait(p_get->ok_to_get());
```

```
            if(!p_put-> nb_ checkwrite (data_size_out_softbits))
                      tm_wait(p_put->ok_to_put());
         }
      }
         //consume some time before gets the input token
         tm_consume(delay_before_get);
         p_get->nb_get(data);
         //consume some time for processing the token
         tm_consume(processing_delay);
         //consume some time before puts the output token.
         tm_consume(delay_before_put);
         data_out= process(data);
         p_put->nb_put(data_out);
         count++;
         if (counter == data_size_in_softbits)
                      counter=0;
   }
}
```

Table 4-5 Clustered task model

There is a counter in clustered task model to record the number of tokens it has processed. Suppose the numbers of input tokens and output tokens in a block are *data_size_in_softbits* and *data_size_out_softbits* respectively. The task starts with checking whether all the input tokens are ready and the output space is big enough. If no, the task constantly puts itself in the *TM_TASK_WAITING* state until the input tokens and output space are available. Then it pulls a token from input channel, and processes it, finally produces it. After steaming a token, the counter is updated. In the next iteration, the task is aware that there must be some tokens left in the input channel, it won't check the input channel again, but continues steaming the data. It won't stop the steaming behavior until the counter is equal to the block size (*data_size_in_softbits*), meaning that a block of tokens have been consumed by the task. Then the counter is reset, and the next round block-level processing begins with the checking of input channel and output channel.

4.6 RR Scheduling

We already created a method to derive a combined SV from a set of implementation aware graphs. Every scheduling unit has a RR scheduler responsible for the scheduling of the tasks running on this scheduling unit. In the implementation, every hardware unit of FLORA is modeled as a VPU, which has a local RR scheduler that can decide the execution order of the tasks mapped on it. But the local RR scheduler is only in charge of one VPU. In order to schedule several VPUs which belong to same scheduling unit as a whole, another remote scheduler is implemented as a task running in another VPU named centralized controller. The communication between tasks and remote scheduler happens over *sc_fifo* channel. An example is shown in Figure 4-9. Suppose *De-Interleaver* is a scheduling unit. Two jobs are virtually mapped on it. Job A consists of tasks TA1 and TA2. Job B consists of a single task TB. Job A and B are scheduled by a RR mechanism.

Assume first TA1 is activated by the local RR scheduler. Then it sends a start request to the remote RR scheduler via *sc_fifo*. If there is no task running on *De-Interleaver*, the remote RR scheduler will start TA1. TA1 first checks if the input tokens and output space are available. If not, TA1 will return the control to the remote RR scheduler immediately. If yes, it will continuously stream a block of tokens. After all the tokens are processed by TA1, local scheduler starts TA2. When TA2 is done, it will return the control to the remote RR scheduler. Next TB is activated by local RR scheduler. It will ask for start permission from the remote RR scheduler. If there is no job running, the remote RR scheduler will start TB immediately. After TB is done, it returns the control to the remote RR scheduler.

The remote scheduler guarantees that in the same time, only the tasks of same job can be activated by the local RR scheduler. Local RR scheduler only worries about the execution order and the start time of the tasks inside the same job. Our model is data driven model. If the input tokens of TA2 are not ready, TA2 won't appear in the ready queue. Therefore, the local RR scheduler can guarantee that the tasks of the same job are activated in the correct order.

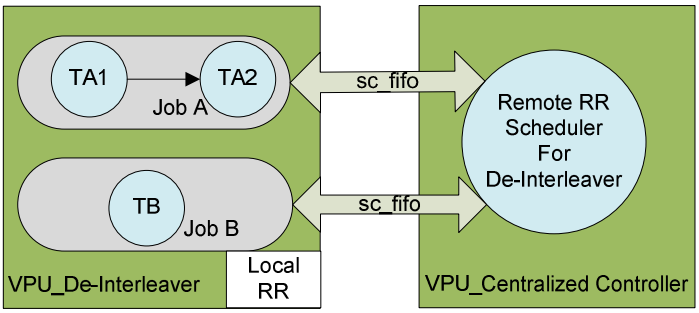

Figure 4-9 Scheduling scheme

The communication between the task and the remote RR scheduler needs to be specified explicitly as shown in Table 4-6. The task starts with sending start request to remote RR scheduler. It won't start until the start command is given by remote RR scheduler. Once the task gets the processor, it checks whether a block of input tokens and the space for a block of output tokens are available. If not, it will return the control of VPU to remote RR scheduler. If yes, it will jump out of the loop, and starts steaming the tokens. After a block of tokens is processed, the task resets the counter to 0, and notifies remote scheduler that it is done.

The remote RR scheduler is implemented as task running on a VPU. The main function is depicted in Table 4-7. The scheduler responds to the start request from the tasks. It activates the job in a RR mechanism.

```
virtual void task(){
while(true){
    if(count == 0){
         while(true){
               //start request
               start_put->nb_put(1);
               //Waiting for the remote scheduler to start the application
               while(!start_put->nb_can_put())
                      tm_wait(start_put->ok_to_put());
```

```
                //Check whether there is input.
                if(p_get->nb_checkread(data_size_in_softbits)&&p_put-
                >nb_checkwrite(data_size_in_softbits))
                {
                        //if everything is ready, jump out of the loop, start the
                        //application
                        break;
                }
                //Otherwise, return the control to Scheduler
                tm_consume(2);
                while(!end_put->nb_can_put()){
                         tm_wait(end_put->ok_to_put());
                }
                end_put->nb_put(1);
            }
     }
    //consume some time before gets the input token
    tm_consume(delay_before_get);
    p_get->nb_get(data);
    //consume some time for processing the token
    tm_consume(processing_delay);
    //consume some time before puts the output token.
    tm_consume(delay_before_put);
    data_out= process(data);
    p_put->nb_put(data_out);
    count++;
    if(count== data_size_in_softbits){
          count=0;
          while(!end_put->nb_can_put()){
                tm_wait(end_put->ok_to_put());
          }
           //if any of input, output space or back pressure is not available,
           //return  the control to Scheduler
          end_put->nb_put(1);
     }
   }
}
```

Table 4-6 Task model communicating with RR scheduler

```
virtual void task(){
        while(true){
             for(i=intlev_running_app_pt_list.begin();i!=intlev_running_app_pt_list
             .end();i++){
                    // start the job
                    if((*(i->start_get_pt))->nb_can_get()){
                         (*(i->start_get_pt))->nb_get(tem);
                        //waiting for the signal that the tasks is done.
                        while(!(*(i->end_get_pt))->nb_can_get())
                                     tm_wait((*(i->end_get_pt))->ok_to_get());
                         (*(i->end_get_pt))->nb_get(tem);
                    }
             }
             tm_wait(1);
        }
}
```

Table 4-7 Remote RR scheduler

4.7 Resource Manager

The resource manager (RM) is employed to do the admission control during run time. A new job can be activated by the end user at any time. Once it is activated, it will send its resource consumption information to RM. The RM has a function called *check_resource()*. The *check_resource* function takes resource consumption model of new job as the input, calculates the throughputs of all the jobs. If all the throughput constraints can be met, the new job is accepted. The RM will inform the remote RR schedulers to add the new job into its RR list. In our implementation, there are 2 remote RR schedulers responsible for De-Interleaver scheduling unit and Output scheduling unit respectively. The RM in Table 4-8 notifies both schedulers to add a new job.

```
void RM::task() {
     while(true){
        if(app_request_get->nb_can_get()){
            app_request_get->nb_get(new_app);
            if(check_resource(new_app)){
                 allocate_resource(new_app);
                 //tell the schedulers that new app are added to system.
                 while(!add_intlev_app_put->nb_can_put())
                        tm_wait(add_intlev_app_put->ok_to_put());
                 add_intlev_app_put->nb_put(new_app);
                 while(!add_output_app_put->nb_can_put())
                        tm_wait(add_output_app_put->ok_to_put());
                 add_output_app_put->nb_put(new_app);
            }
            else{
                   cout<<sc_time_stamp() <<",   We can't schedule this application
                   with job_id:"<<(int)new_app.job_id <<endl;
            }
        }
    }
}
```

Table 4-8 RM example

check_resource function calculates the throughputs of the jobs based on the MCM of the HSDF graphs and the WCWT of the tasks. If the throughputs of all the jobs meet the requirements, the function will return true. Otherwise it will return false.

The overview of the simulation system is shown in Figure 4-10.

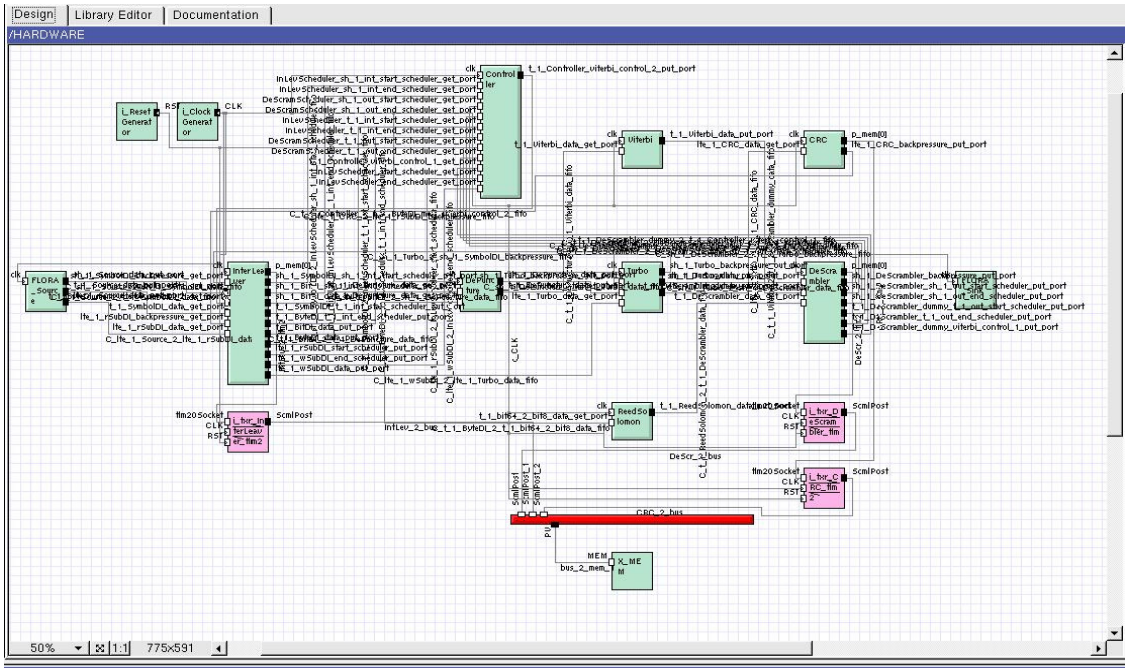

Figure 4-10 Overview of simulation system

5 Results

5.1 Introduction

To verify the approach, we created the task models for DVB-T, DVB-SH, and LTE in Coware ESL2.0. These task models are mapped onto the simulation platform of FLORA. The matrix inner connections of FLORA are configured to match the connections required by the applications. We can also calculate the combined SV of these 3 applications. The SV and routing are shown in Figure 5-1. There are 4 scheduling units, from SU1 to SU4. Each of them has an online RR scheduler, running on the centralized controller. The communication between scheduling units and the remote RR scheduler happens over *sc_fifo* (dash line in Figure 5-1). Besides the online RR schedulers, a RM is also created in centralized controller, to perform the admission control.

We want to emphasize that these applications do not share the buffers of the Turbo decoder. Instead, each of them has its own buffers of the Turbo decoder, such that these applications won't depend on each other. For instance, LTE and DVB-SH both are using Turbo decoder. If they are sharing Turbo input buffer, the tasks of LTE can't start filling Turbo input buffer until the data of DVB-SH application in Turbo input buffer are emptied. This delays the execution of the LTE, but most importantly, makes the application highly depends on each other. To simply our scheduling method, we allow the application to have its own Turbo buffers.

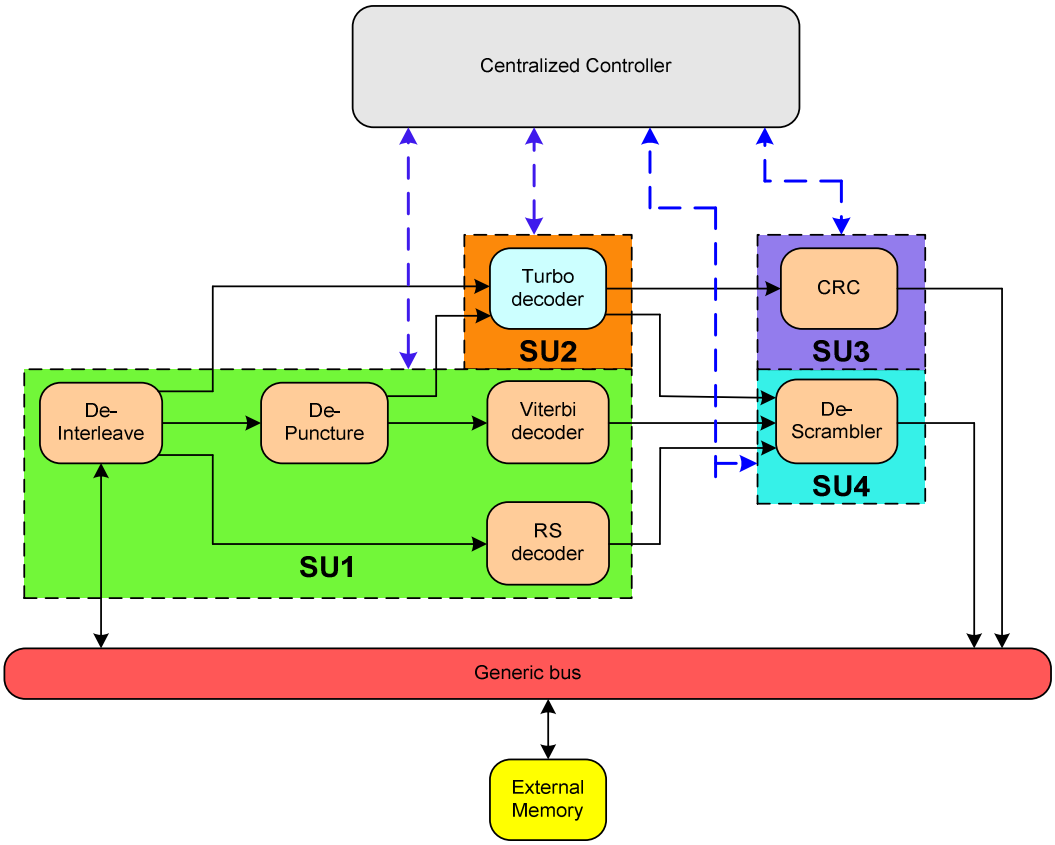

Figure 5-1 the routing and the combined SV

In the simulations, we want to show how the RR scheduler and RM handle the dynamic mix of several radios. Several scenarios will be simulated:

- DVB-T and DVB-SH are mapped on FLORA and scheduled by online RR schedulers.
- DVB-T, DVB-SH, and LTE are mapped on FLORA and scheduled by online RR schedulers without RM.
- DVB-T, DVB-SH, and LTE are mapped on FLORA and scheduled by online RR schedulers with RM.

5.2 Dynamic mix of DVB-T and DVB-SH

Before we explain the result of the simulation, let's first look at Table 5-1. The source of DVB-T produces a block of tokens every 800us. For DVB-SH, the source produces a block of tokens every 400us. In the real implementation, the heart beat for both DVB-T and DVB-SH is 924us. The reason we tune the frequency of sources higher than 1/924us is that we want increase the workload FLORA and reduce its idle time. This won't change the WCRT of tasks or the worst case throughput of the application. The throughput constraints are same for both applications, which require a block of output tokens every 924us. For DVB-T application, the scheduling units SU1 and SU4 are coupled. For DVB-SH application, the scheduling units SU1, SU2 and SU4 are working independently from each other.

		SU1	SU2	SU3	SU4
DVB-T Source period: 800us Throughput: 1/924us Mode: 8K, 64QAM	Virtual Mapping	*t_1_SymbolDI* *t_1_BitDI* *t_1_Depunctuer* *t_1_Viterbi* *t_1_ByteDI* *t_1_ReedSolomon*			*t_1_Descrambler_dummy* *t_1_DeScrambler*
	Execution Time	300us			186us
DVB-SH Source period: 400us Throughput: 1/924us 8K, 16QAM CR: N=9	Virtual Mapping	*sh_1_SymbolDI* *sh_1_TimeDI* *sh_1_BitDI*	*sh_1_Turbo*		*sh_1_Descrambler*
	Execution Time	180us	108us		10us

Table 5-1 DVB-T and DVB-SH virtual mapping and execution times

Based on the analysis model of DVB-T and DVB-SH (please see Appendix), it is proved that there are enough resources on FLORA to run DVB-T and DVB-SH in parallel as shown in Table 5-2. The MCMs for DVB-T and DVB-SH are smaller than 924us. In the first simulation, we will run DVB-T and DVB-SH under RR schedulers to show how the RR scheduler works. The simulation result is shown in Figure 5-2 and Figure 5-3.

	SU1		SU2		SU3		SU4		MCM
	WCWT	WCET	WCWT	WCET	WCWT	WCET	WCWT	WCET	
DVB-T	180us	300us	0	0	0	0	10us	186us	490us
DVB-SH	310us	180	0us	108us	0	0	186us	10us	490us

Table 5-2 Schedulability analysis for DVB-T and DVB-SH

The overall executions of DVB-T and DVB-SH are shown in Figure 5-2. With the online RR scheduler, the application can be activated and stopped dynamically. The DVB-SH and DVB-T are both activated at very beginning. At around 800us of the time line, DVB-SH is stopped. The DVB-T application owns all the resources of FLORA. At around 3.4ms, DVB-SH is activated again. The RR schedulers add the DVB-SH to their scheduling lists. There is also inter-jobs parallelism among different scheduling units. At 4.9ms, SU1 is busy with DVB-T and SU2 is working on DVB-SH.

t_1_Descrambler in red bubble B1 and *sh_1_ Descrambler* in blue bubble B2 are producing output data for DVB-T and DVB-SH respectively. Following the timeline, after 4.9ms, they produce the output data periodically. Arrow 1 and arrow 2 indicate the maximal gap between 2 continuous output blocks. They are smaller than 924us. In this simulation graph, the throughput requirements for DVB-T and DVB-SH are met.

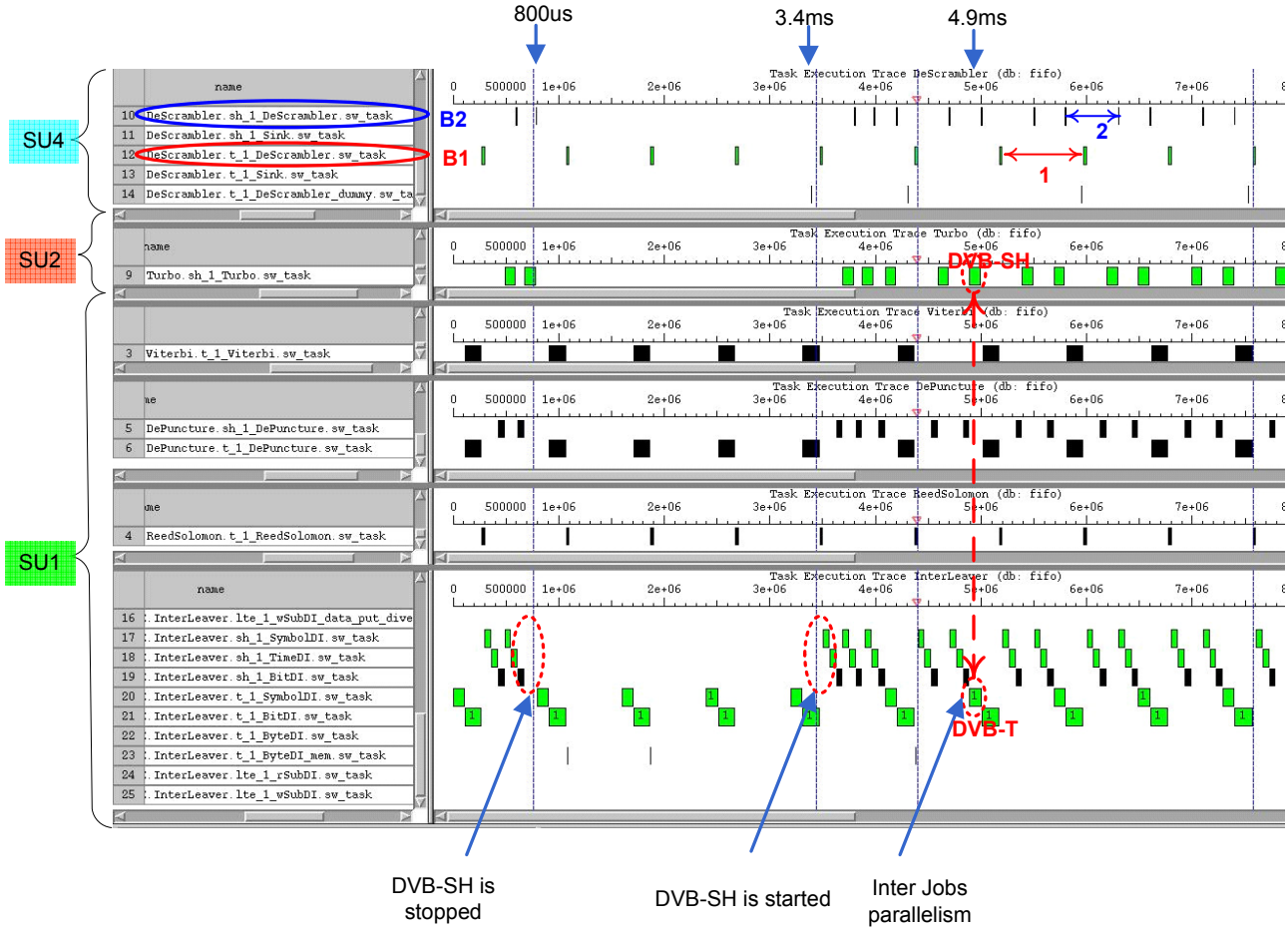

Figure 5-2 Dynamic mix of DVB-T and DVB-SH

If we zoom in on the first 700us, the execution trace will be the one shown in Figure 5-3. At the beginning, DVB-T and DVB-SH are activated. The RR scheduler of scheduling unit SU1 selects the DVB-T application and starts the application.

The streaming-level parallelism inside the same scheduling unit can be seen in Figure 5-3. For example, from 100us to almost 300us, *t_1_ByteDI, t_1_Depuncture and t_1_Viterbi* are streaming the data at small granularity simultaneously. Although these 3 tasks are virtually mapped on the same scheduling unit SU1, the hardware units inside SU1 are working in the streaming level pipeline.

At about 300us, DVB-T is done. The RR scheduler on SU1 immediately starts DVB-SH.

There is block-level pipeline among SU1, SU2 and SU4. For instance, at the 500us of the timeline, SU1 and SU2 are working in a pipeline mode where SU2 is busy with the execution of last round of *sh_1_Turbo*, and SU1 is busy with *sh_1_SymbolDI and sh_1_TimeDI* in current round.

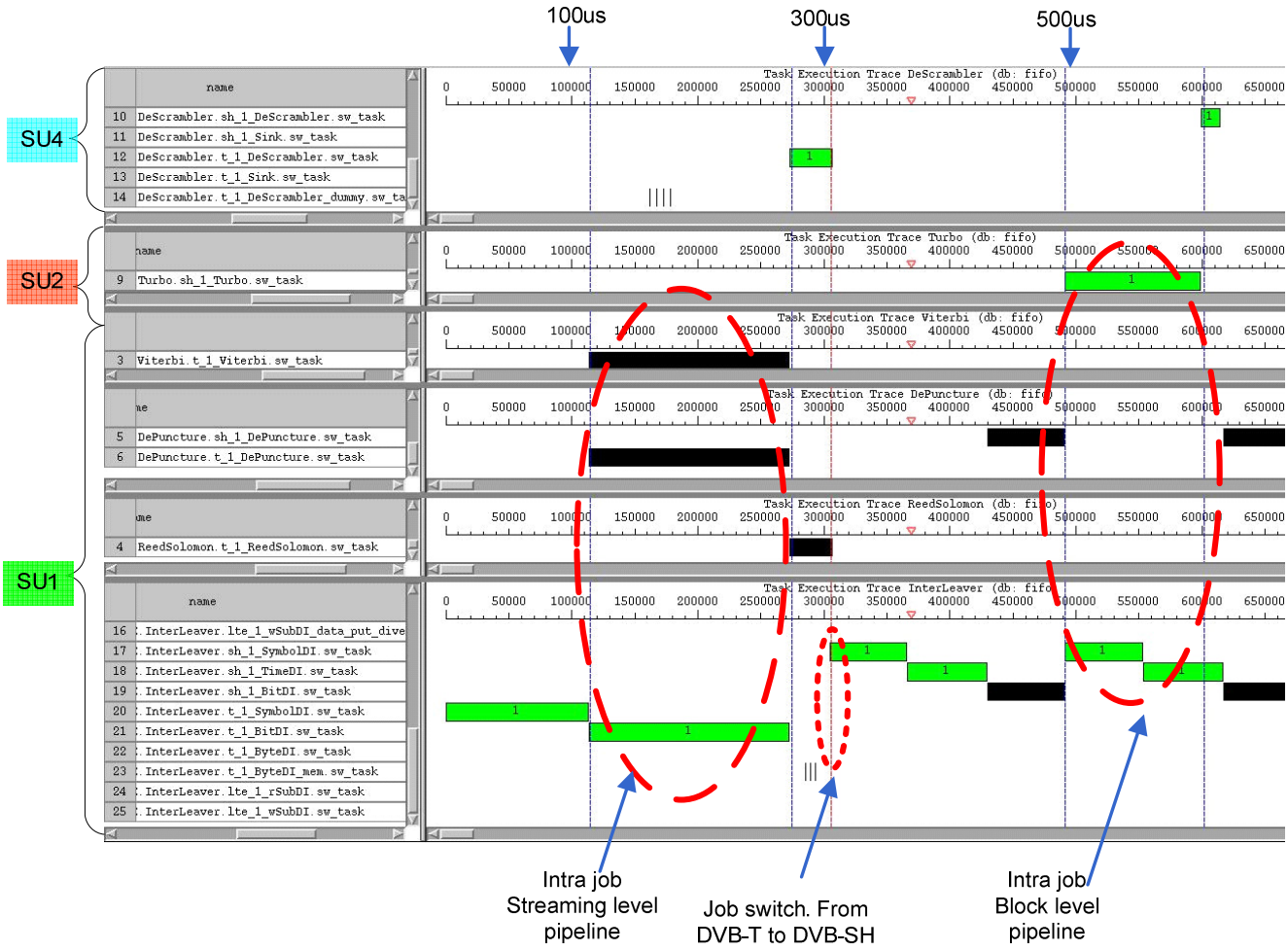

Figure 5-3 Starting phase of DVB-T and DVB-SH

In this simulation, 2 applications are sharing the FLORA. They are scheduled in RR mechanism. There are enough resources for both DVB-T and DVB-SH on FLORA. The combination of them can be handled by the RR schedulers and they can meet the throughput constraint. There is no deadlock, missing deadline or starvation.

5.3 Dynamic mix of DVB-T, DVB-SH and LTE without RM

In this section, we want to simulate a dynamic mix of DVB-T, DVB-SH, and LTE. We want to emphasize that FLORA was designed for broadcasting standards such as DVB-T and DVB-SH. Therefore, the buffer sizes in FLORA are tailored for these applications. For the cellular standard like LTE, we are still in the exploration phase. The buffer size of each hardware unit for LTE channel is not decided yet. Different buffer size can result in different scheduling methods.

According to the buffer size of Turbo decoder, the LTE application can be scheduled in 2 ways. In the implementation independent SDF graph of LTE in Figure A. 4 (please see Appendix), for each firing, the source produces a transport block (TB), which consists of 13 code blocks (CB). Functionally, the tasks such as *SubDI*, *Turbo* and *CRC* consume one CB per firing. If we set the input buffer size of *Turbo* to be 1 CB, then the *SubDI* and *Turbo* are processing data at CB level. *SubDI* and *Turbo* can be synchronized by coupled scheduling to finish processing a TB (13 CBs) data together. The buffer size is reduced to one CB, but the coupled scheduling will consume extra time to synchronize *SubDI* and *Turbo* and increases the workload of SU1. If we set the input buffer size of *Turbo* to be 1 TB (13 CBs), the *SubDI* can continuously process all 13 CBs and then store the 13 CBs to the input buffer of *Turbo*. Next *Turbo* continuously processes all 13 CBs. No coupled scheduling is needed, but the buffer size is increased. The implementation aware graphs of both cases are shown in Appendix. The virtual mapping and execution times for both cases are shown in the Table 5-3.

		SU1	SU2	SU3	SU4
LTE Turbo with a TB size buffer Source period: 1ms Throughput: 1/1ms Category: 4, 20MHz	Virtual Mapping	*lte_1_rSubDI* *lte_1_wSubDI*	*lte_1_Turbo*	*lte_1_CRC*	
	Execution Time	377us	600us	100us	
LTE Turbo with a CB size buffer Source period: 1ms Throughput: 1/1ms Category: 4, 20MHz	Virtual Mapping	*lte_1_rSubDI* *lte_1_wSubDI*	*lte_1_Turbo*	*lte_1_CRC*	
	Execution Time	600us	600us	100us	

Table 5-3 LTE virtual mapping and execution times

5.3.1 LTE without coupled scheduling

We first do a simulation of the dynamic mix of DVB-T, DVB-SH and LTE. The input buffer size of the Turbo decoder is 1 TB. RM is not assigned.

The simulation result is shown in Figure 5-4. SU1 and SU2 are not coupled in this simulation. This can be seen from bubble B1 and Bubble B2 in Figure 5-4. *lte_1_rSubDI* and *lte_1_wSubDI* stream 13 CBs to the input buffer of SU2 (Turbo decoder). After the buffer is full, SU2 continuously processes all the CBs. SU1 and SU2 are not coupled. They are scheduled independently.

Based on the MCM calculation of analysis models (please see appendix), it shows that LTE can't be scheduled with DVB-T or DVB-SH by this scheduling method as shown in Table 5-4 and Table 5-5. The MCMs for LTE in both cases are bigger than 1000us. LTE can't meet throughput requirement. There are several indications that can be found in Figure 5-4. The task *lte_1_CRC* is virtually mapped on SU3, which produces the output of LTE application. If we follow the timeline, before the DVB-SH is started, *lte_1_CRC* produces an output block every 1ms (The time interval of arrow 1), which just meets throughput requirement of LTE. At 3ms, DVB-SH is added to system. The period of producing a block of output data for *lte_1_CRC* is increased to more than 1ms, as indicated by arrow 2. After DVB-T is started at around 5ms, it takes more time for LTE to produce the output data. There are even no output tokens produced by *lte_1_CRC* from 8ms to 9ms (in the position of the red question mark).

	SU1		SU2		SU3		SU4		MCM
	WCWT	WCET	WCWT	WCET	WCWT	WCET	WCWT	WCET	
DVB-T	377us	300us	0	0	0	0	0	186us	677us
LTE	300us	377us	0us	600us	0	100us	0	0	1277us

Table 5-4 Schedulability analysis for DVB-T and LTE without coupled scheduling

	SU1		SU2		SU3		SU4		MCM
	WCWT	WCET	WCWT	WCET	WCWT	WCET	WCWT	WCET	
DVB-SH	377us	180	600us	108us	0	0	0	10us	1157us
LTE	180us	377us	108us	600us	0	100us	0	0	1265us

Table 5-5 Schedulability analysis for DVB-SH and LTE without coupled scheduling

From the simulation combined with the calculation from the analysis models, we come to conclusion that LTE can't be scheduled with DVB-T or DVB-SH by our approach if the input buffer size of Turbo is 1 TB. The potential solution is to double the input buffer size of Turbo. That will reduce the MCM and improve the throughput of LTE. But on-chip memory is expensive. 2-TB buffer can be around 450K bytes. The trade-off has to be made between them.

Without the RM doing the admission control, the applications that are dynamical mixed don't have the guarantee for their temporal behaviors.

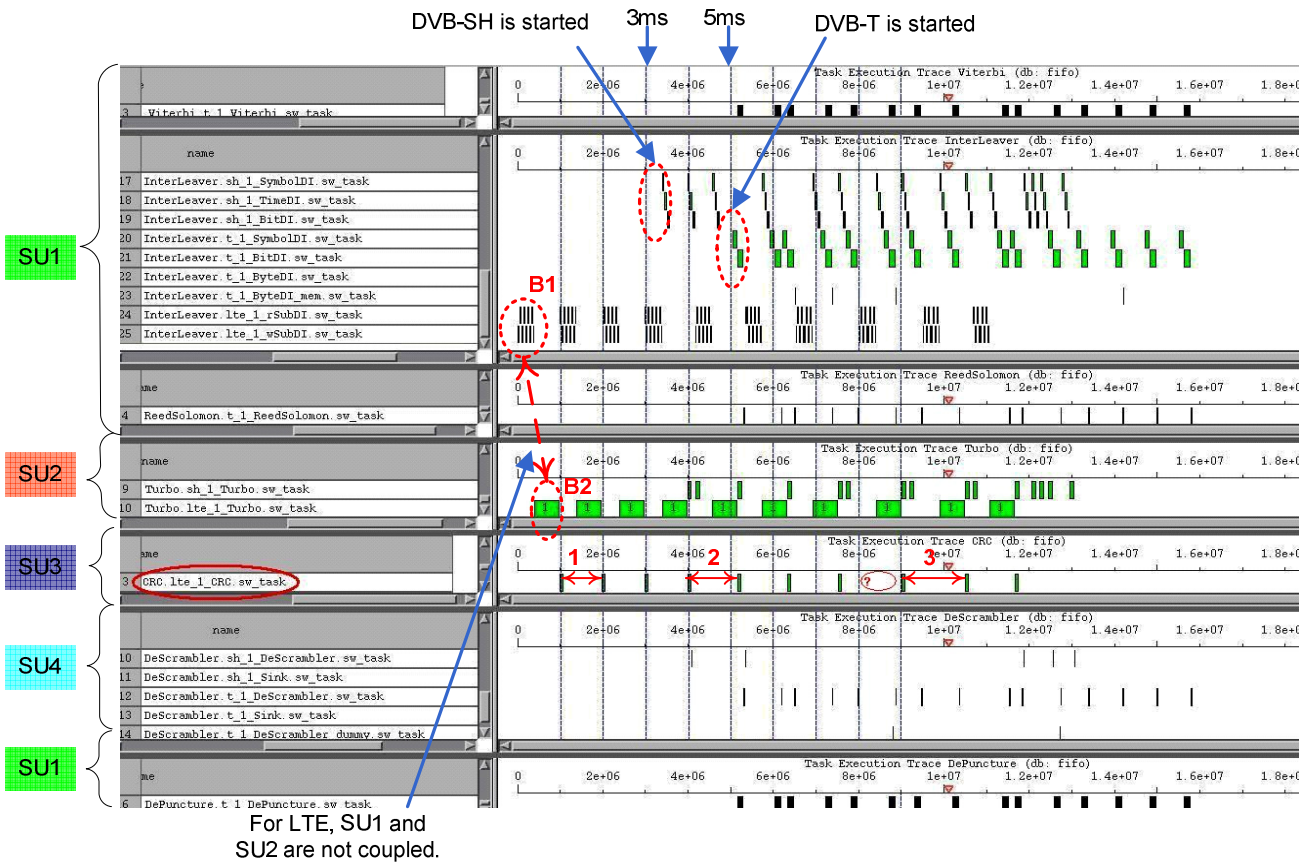

Figure 5-4 Dynamic mix of DVB-T DVB-SH and LTE without coupled scheduling

5.3.2 LTE with coupled scheduling

This simulation is also about the dynamic mix of DVB-T, DVB-SH and LTE. The input buffer size of the Turbo decoder is 1 CB. The coupled scheduling is used in this simulation. RM is not assigned. Based on the MCM calculation of analysis model (please see appendix), it shows that LTE can be scheduled with DVB-T but not DVB-SH by this scheduling method, as shown in Table 5-6 and Table 5-7. For the combination of DVB-T and LTE, the MCMs for them just meet the throughput requirement. For the combination of DVB-SH and LTE, LTE just meets throughput requirement, but DVB-SH can't.

	SU1		SU2		SU3		SU4		MCM
	WCWT	WCET	WCWT	WCET	WCWT	WCET	WCWT	WCET	
DVB-T	600us	300us	0	0	0	0	0	186us	900us
LTE	300us	600us	0us	600us	0	100us	0	0	1000us

Table 5-6 Schedulability analysis for DVB-T and LTE with coupled scheduling

	SU1		SU2		SU3		SU4		MCM
	WCWT	WCET	WCWT	WCET	WCWT	WCET	WCWT	WCET	
DVB-SH	708us	180	600us	108us	0	0	0	10us	1488us
LTE	180us	600us	108us	600us	0	100us	0	0	998us

Table 5-7 Schedulability analysis for DVB-SH and LTE with coupled scheduling

The simulation result is shown in Figure 5-5 and Figure 5-6. Same as previous simulation, LTE is started at the beginning. At around 3.5ms, DVB-SH is injected to the system. The RR schedulers of the scheduling units decide the execution order and start time of all the tasks. At around 6.9ms, DVB-T is added to system. Now 3 applications are running together. Based on the worst case throughput calculation from analysis models, LTE can't be scheduled with both DVB-T and DVB-SH. But due to the short simulation time, this is not shown in Figure 5-5.

The coupled scheduling between SU1 and SU2, as an example, can be seen from task execution traces inside bubble B1 and bubble B2. The LTE tasks *lte_1_rSubDI* and *lte_1_wSubDI* are coupled with *lte_1_Turbo*. They process the data at the CB level pipeline and steam a whole TB (13 CB) data to next stage.

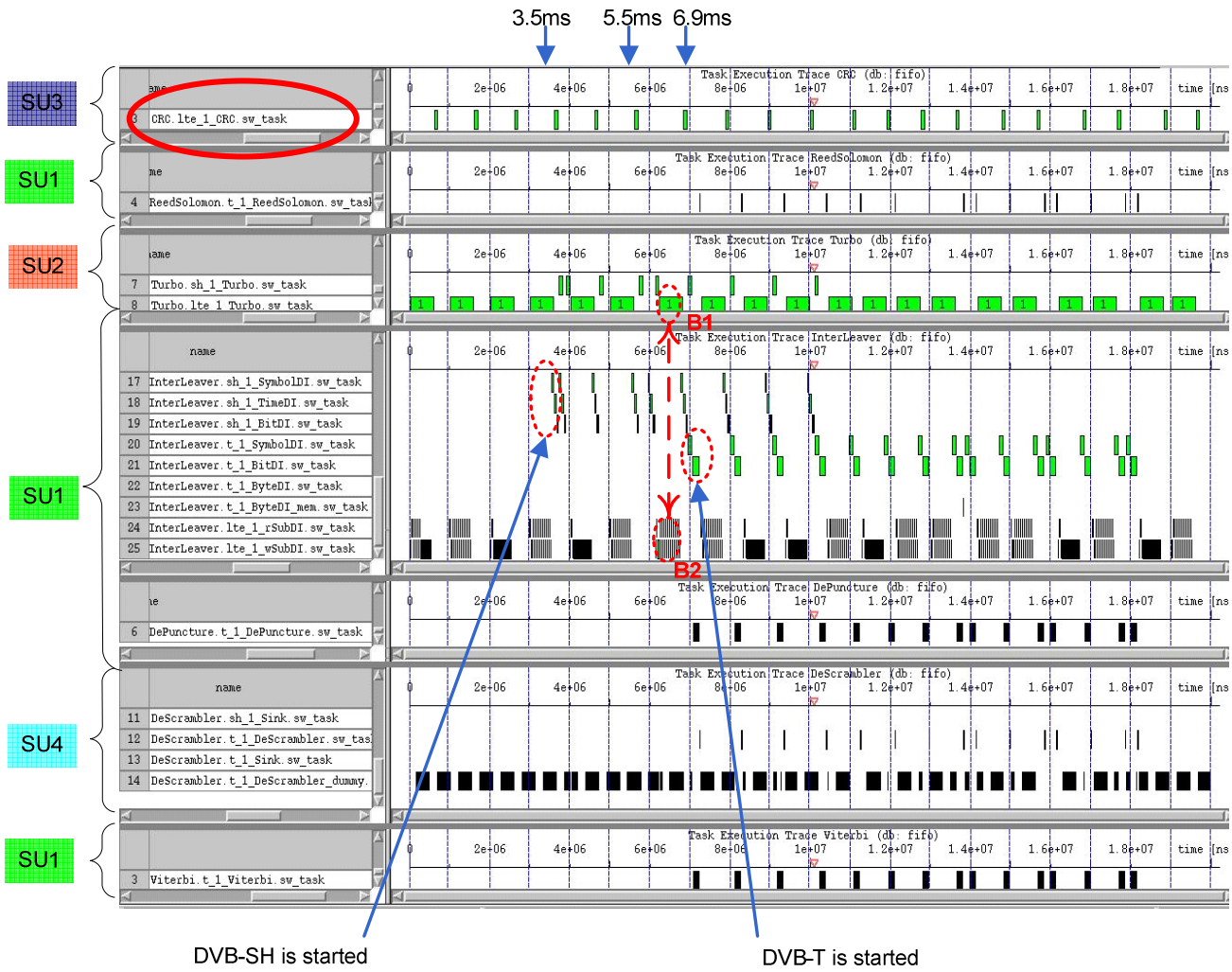

Figure 5-5 Dynamic mix of DVB-T DVB-SH and LTE with coupled scheduling

The coupled scheduling is more obvious if we zoom into the time period from 5.5m to 7.4ms, as shown in Figure 5-6. It is noticeable that execution trace in bubble C2 is slightly different from its neighbors: the ones in bubble C3 and bubble C4. If we look vertically, analyze bubble C2 and C1 together, we will find the reason. The tasks in bubble C1 is *sh_1_Turbo*, which belongs to DVB-SH. During a small period, SU2 is occupied by *sh_1_Turbo*. For LTE application, SU1 and SU2 are coupled. When the tasks of LTE in bubble C2 are holding the processing resources of SU1, they still have to wait until *sh_1_Turbo* in bubble C1 is done. This is the cost of coupled scheduling.

Not only this, in the Table 5-3, LTE with coupled scheduling takes more execution time in SU1 compared with LTE without coupled scheduling. This is because of the coupled scheduling. The coupled scheduling forces SU1 to slow done, since SU2 can't process the data as fast as SU1.

From the simulation combined with the calculation from the analysis models, we come to conclusion that LTE with coupled scheduling can be scheduled with DVB-T, but not DVB-SH. The coupled scheduling for LTE needs smaller input buffer of Turbo decoder. However, LTE with coupled scheduling can't be scheduled with DVB-SH, since DVB-SH can't meet is throughput requirement in the worst case scenario. The coupled scheduling brings synchronization cost between SU1 and SU2. If the other application has a big task in SU2, it will be a waste that the LTE tasks in SU1 is doing nothing but just waiting for SU2. The coupled scheduling also slows done the SU1 because that the SU2 consumes more time to process a CB data.

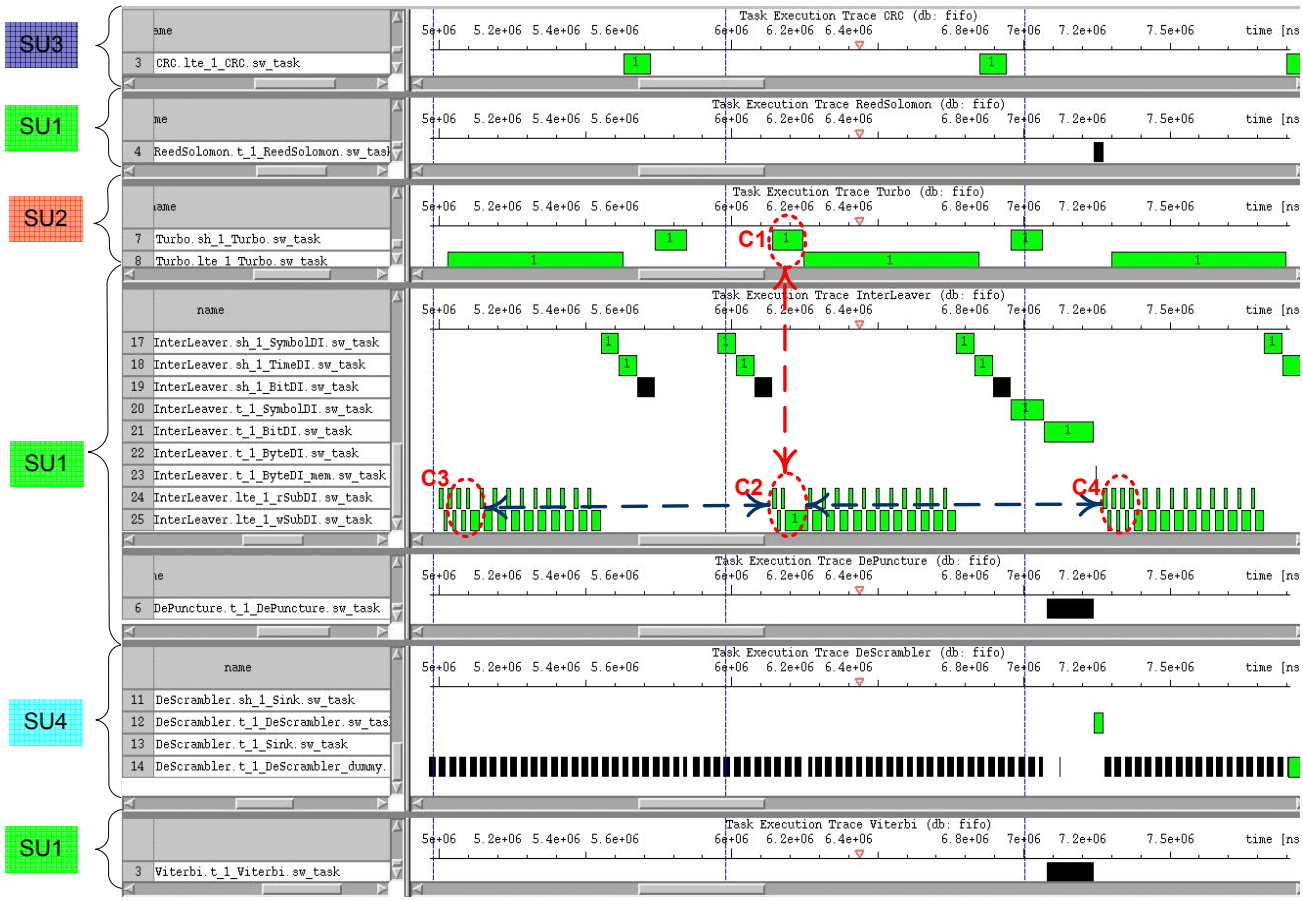

Figure 5-6 zoom-in graph of Dynamic mix of DVB-T DVB-SH and LTE with coupled scheduling

5.4 Dynamic mix of DVB-T, DVB-SH and LTE with RM

In this simulation, for LTE, we set the input buffer for Turbo to be a TB. No coupled scheduling is imposed on LTE. RM is added to the simulation platform. The result is shown in the Figure 5-7.

The LTE application starts execution from the very beginning. In the middle, DVB-SH application sends the start request to RM. The RM checks if there are enough resources by calculating the CMC of the analysis models for LTE and DVB-SH. It turns out LTE won't meet throughput requirement if DVB-SH is added. Therefore, RM refuses the request of DVB-SH. At around 9.5ms, LTE is done, and then DVB-SH asks for the start. RM calculates the CMC of analysis model for DVB-SH again. The result is that DVB-SH can meet its throughput requirement. Then RM informs the RR schedulers. The RR schedulers add DVB-SH in there scheduling lists and DVB-SH starts execution. Later on, at around 11ms, the DVB-T also asks for permission from RM. The RM checks the resources again, and asks the RR schedulers to add the DVB-T application in their scheduling lists. The DVB-T is started and running with DVB-SH simultaneously.

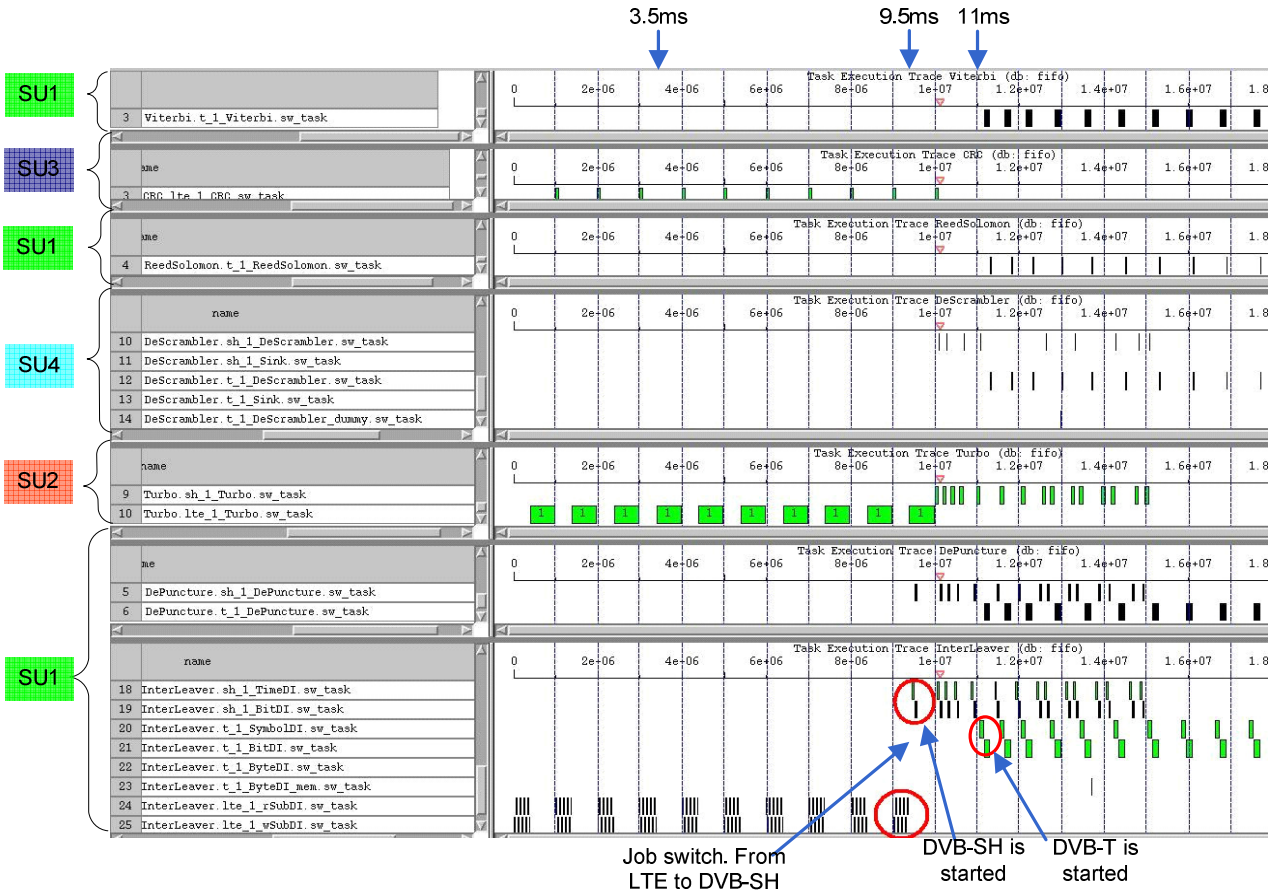

Figure 5-7 Dynamic mix of DVB-T, DVB-SH and LTE with RM

6 Conclusion and Future Work

In this thesis, we present a design flow for scheduling hard-real-time applications with a dynamic job-mix on FLORA. The scheduling consists of two parts: compile-time scheduling and run-time scheduling.

During compile time, an implementation independent SDF graph for each radio is created according to the radio's specification. Next, an implementation aware SDF is derived from the implementation independent SDF graph combined with the hardware mapping information. Then we developed a method to derive a combined SV, which is an optimal hardware partition for the scheduling ability of FLORA, from a set of implementation aware SDF graphs. Furthermore the implementation aware graph is virtually mapped on the scheduling units of the SV. Clustering, coupled scheduling and RR scheduling are applied to the actors in the SDF graph after virtual mapping. In the end, an analysis SDF model for reach radio is generated.

During run time, the RR schedulers and RM are added. The RR scheduler accounts for scheduling the tasks mapped on a scheduling unit. Dynamic start and stop of jobs can happen at run time. In order to guarantee the resource provision for the running jobs and the new coming job, a RM is designed. The RM checks the availability of current resources, and calculates the MCM from the analysis SDF model of each job. If the admission rules can be met, the new job will be accepted and scheduled. If not, the new job will be denied.

A simulation system is built to verify our approach. We choose the Coware ESL tool to setup the simulation system. The Coware task models for DVB-T, DVB-SH and LTE are created. We applied compile time scheduling techniques to these task models. Furthermore, a simulation hardware platform for FLORA is also built. The task models after compile time scheduling, are mapped on the simulation platform. Based on various mappings, we demonstrated how the run-time RR schedulers and RM are working in the final results.

With the results as the proof, the research objective that was formulated in 0 is considered to be sufficiently answered.

Although the dynamic combination of several radios can be handled by FLORA now, there are still some issues deserve further research. These include:

- We didn't prove that the algorithm used to derive the combined SV from a set of single-iteration SVs is deterministic. If it is not, the combined SV will depend on the sequence of the calculation. This may result in several combined SVs for a set of applications.

- Clustering can only combine the actors and estimate the execution time of the new combined actor under very strict conditions. The SDF graph of an application can be more complex such that it can't meet strict conditions for clustering. For instance, the implementation aware graph of DVB-SH in Figure 3-5 has a cyclo-static expression for *De-puncture*. For these complex graphs, we can't calculate accurate execution times for each actor by the techniques mentioned in the clustering step. Instead, in this thesis, most execution times of the tasks after clustering are based previous experiments.
- The coupled scheduling brings synchronization cost. In some worst scenario, the synchronization cost can affect the performance of all the applications.
- The implementation of the online schedulers of the simulation system is not well formed. This can be improved by using standard Coware API. Coware has a better pre-defined API for scheduling. This is not well used in the implementation.

7 Reference

[1] T. Hentschel, M. Henker, G. Fettweis, The Digital Front-End of Software Radio Terminals, *IEEE Personal Communications Magazine*, 6-12, 1999.

[2] N. Boumaaz, et al. Simplified design for digital front end using random sampling in software defined radio architecture, *Wireless Technology,* 2006

[3] Kees van Berkel et al, Vector Processing as an Enabler for Software-Defined Radio in Handheld devices. *EURASIP Journal on Applied Signal Processing,* (16), 2005.

[4] E. Tell, A. Nilsson, D. Liu, A Programmable DSP core for Baseband Processing, in *IEEE-NEWCAS Conference*, Jun 2005

[5] http://www.design-reuse.com/articles/15703/silicon-ip-for-programmable-baseband-processing.html

[6] http://www.sdrforum.org/pages/aboutSdrTech/whatIsSdr.asp

[7] http://www.arm.com/products/solutions/AMBAHomePage.html

[8] E. Lee and D. Messerschnitt. Synchronous data flow. In *Proceedings of the IEEE*, 1987.

[9] M. Bekooij et al. Dataflow analysis for real-time embedded multiprocessor system design. *In Proc. Int'l Workshop SCOPES*, LNCS 3199. Springer, Sept. 2004

[10] S. Sriram, and S. Bhattacharyya. *Embedded multiprocessors: Scheduling and Synchronization*, Marcel Dekker, Inc., 2000.

[11] Moreira et al. Scheduling multiple independent hard real time jobs on a heterogeneous multiprocessor. *Proceedings of the 7th ACM & IEEE international conference on Embedded software,* 2007.

[12] Moreira and M. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing,* 2007

[13] Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. ACM *Transactions on Design Automation of Electronic Systems*, 9(4):385-418, Oct. 2004.

[14] M. Wiggers, M. Bekooij, and G. Smit. Modelling Run-Time Arbitration by Latency-Rate Servers in Data Flow Graphs. *In Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES), April 2007.*

[15] www.coware.com

[16] Task modeling and virtual processing unit user's guide.

Acronyms and Terms

3G	Third Generation
API	Application Programming Interface
DVB-T	Digital Video Broadcasting - Terrestrial
DVB-SH	Digital Video Broadcasting – Satellite services to Handhelds
ESL	Electronic System Level
FLORA	Flexible outer receiver architecture
HU	Hardware unit
HS	Hardware segment
HSDF	Heterogeneous synchronous data flow
JS	Job structure
LTE	Long term evolution
MC	Micro-controller
OM	Original mapping
RM	Resource manager
RR	Round Robin
RD	Relative deadline
SV	Scheduling view
SDR	Software defined radio
SDF	Synchronous data flow
WCET	Worst case execution time
WCRT	Worst case response time
WCWT	Worst case waiting time

Appendix A: Graphs

A.1 The implementation independent SDF graph

Implementation independent SDF graphs for all the applications are shown as below. For DVB-SH, see page 25.

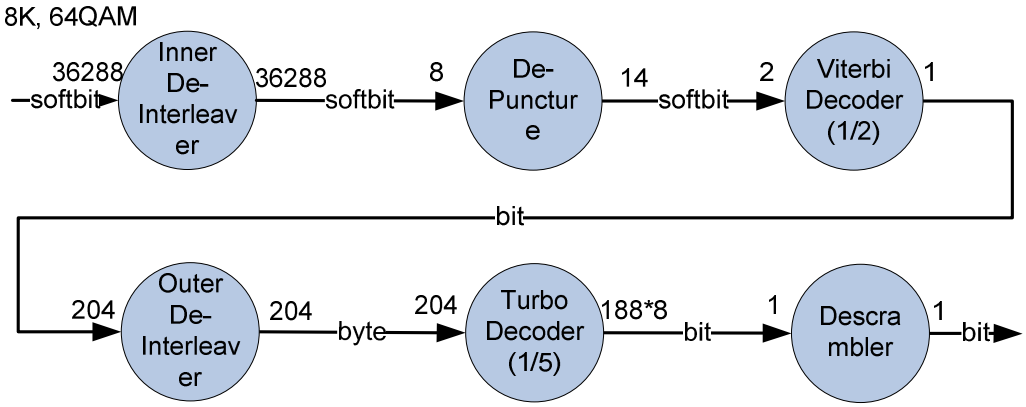

Figure A. 1 Implementation independent SDF graph for DVB-T

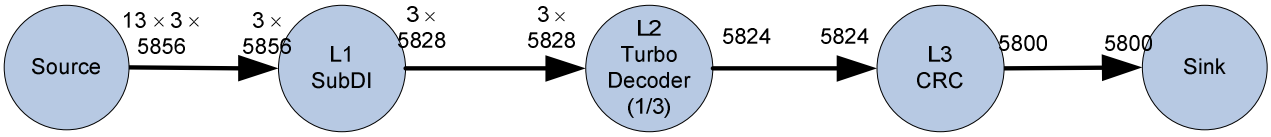

Figure A. 2 Implementation independent SDF graph for LTE

A.2 The implementation aware SDF graph

The implementation aware SDF graphs for all the applications are shown as below. For DVB-SH, see page 26.

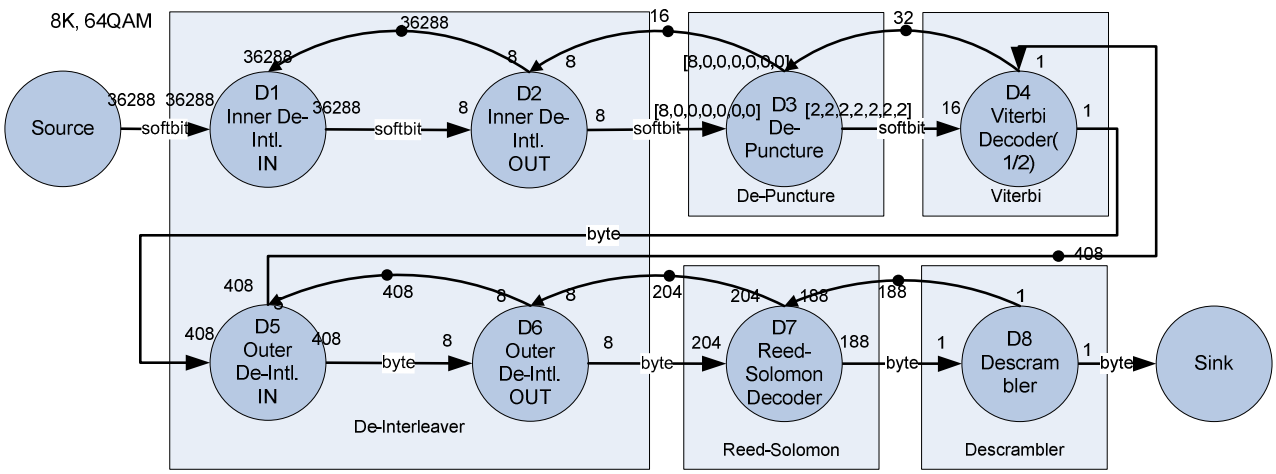

Figure A. 3 Implementation aware SDF graph for DVB-T

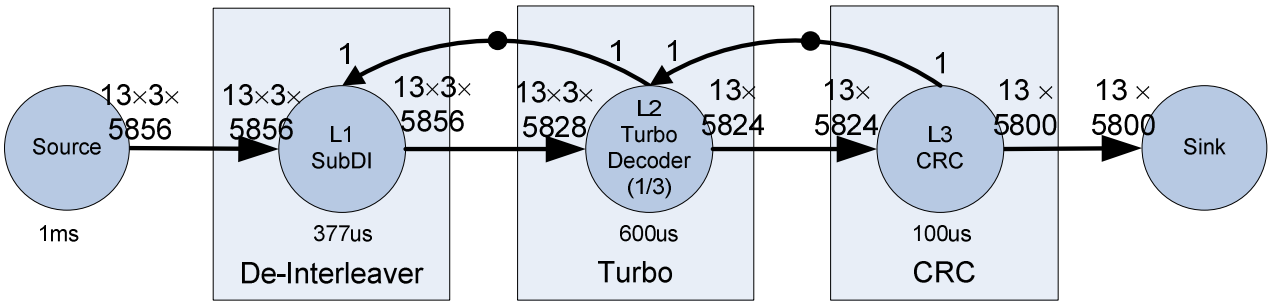

Figure A. 4 Implementation aware SDF graph for LTE with TB sized Turbo input buffer

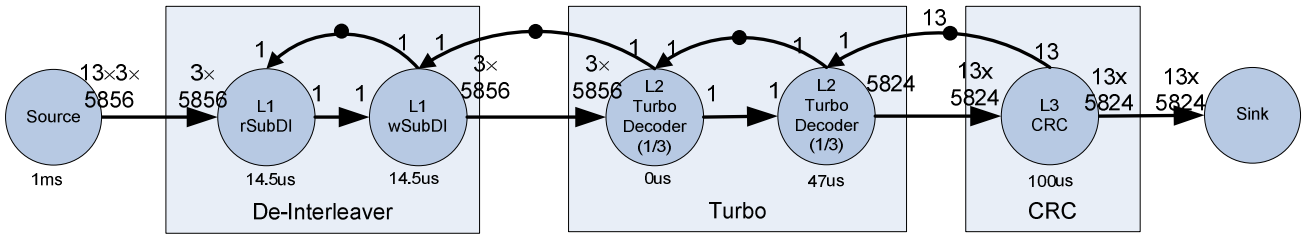

Figure A. 5 Implementation aware SDF graph for LTE with CB sized Turbo input buffer

A.3 The single-iteration SVs and combined SV

Iterations for DVB-T are $[(D1;D2;D3;D4)],[(D5;D6;D7;D8)]$

Single-iteration SVs for DVB-T are
$[(De-Interleaver, Depuncture, Viterbi, Descrambler)]$,
$[(De-Interleaver, ReedSolomon, Descrambler)]$

Iteration for DVB-SH is $[(A1;A2;A3;A4;A5);(A6);(A7);(A8)]$

Single-iteration SV for DVB-SH is $[(De-Interleaver, De-Puncture),(Turbo),(De-Scrambler)]$

Iterations for LTE is $[(L1);(L2);(L3)]$

Single-iteration SV for LTE is $[(De-Interleaver),(Turbo),(CRC)]$

From the single-iteration SVs, we can derive ***combined SV*** for DVB-T, DVB-SH and LTE. It is $[(De-Interleaver, Depuncture, Viterbi, ReedSolomon),(Turbo),(Descrambler),(CRC)]$.

A.4 Analysis model

The Analysis models are shown below. The clustering and coupled scheduling are applied.

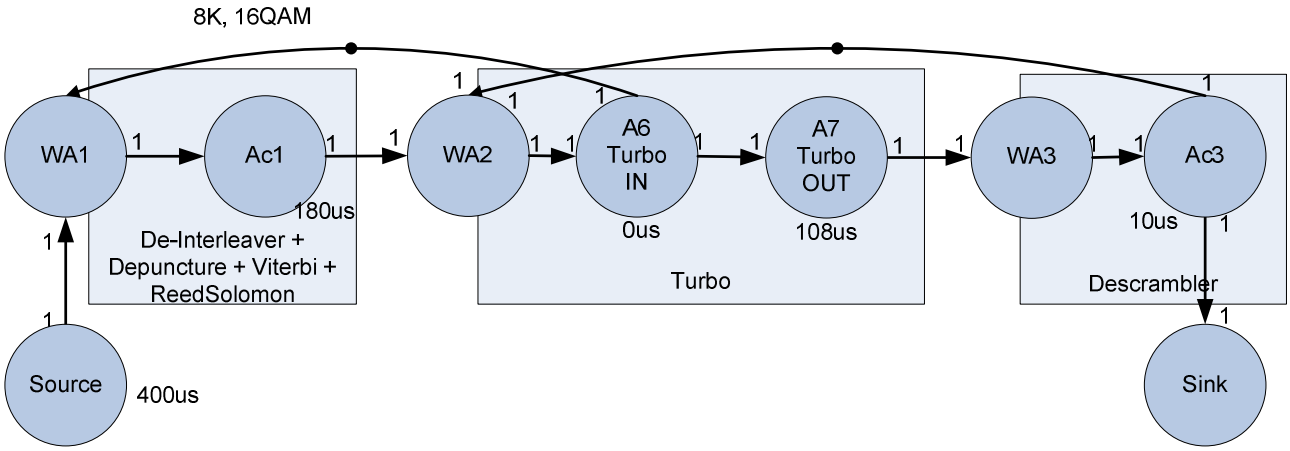

Figure A. 6 Analysis model for DVB-SH

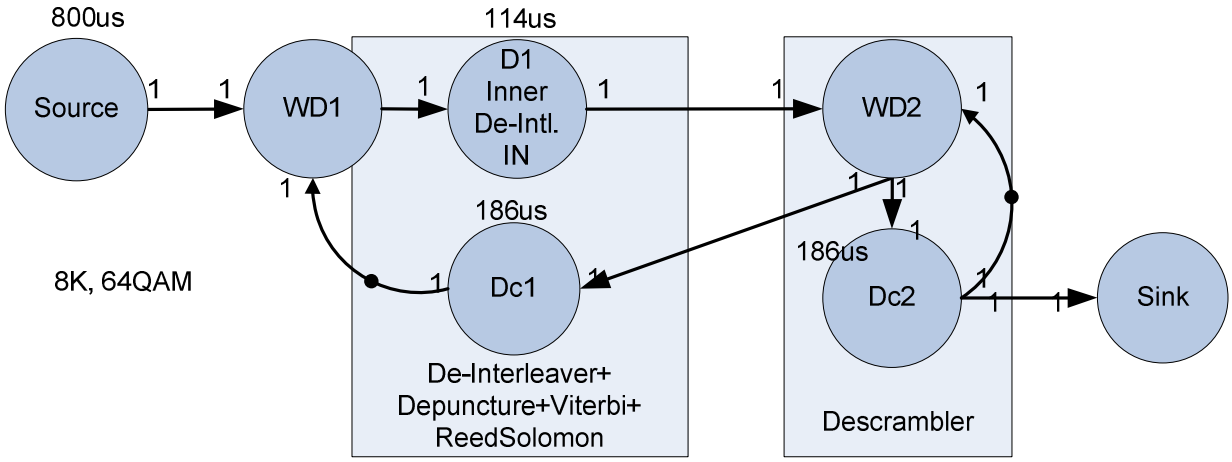

Figure A. 7 Analysis model for DVB-T

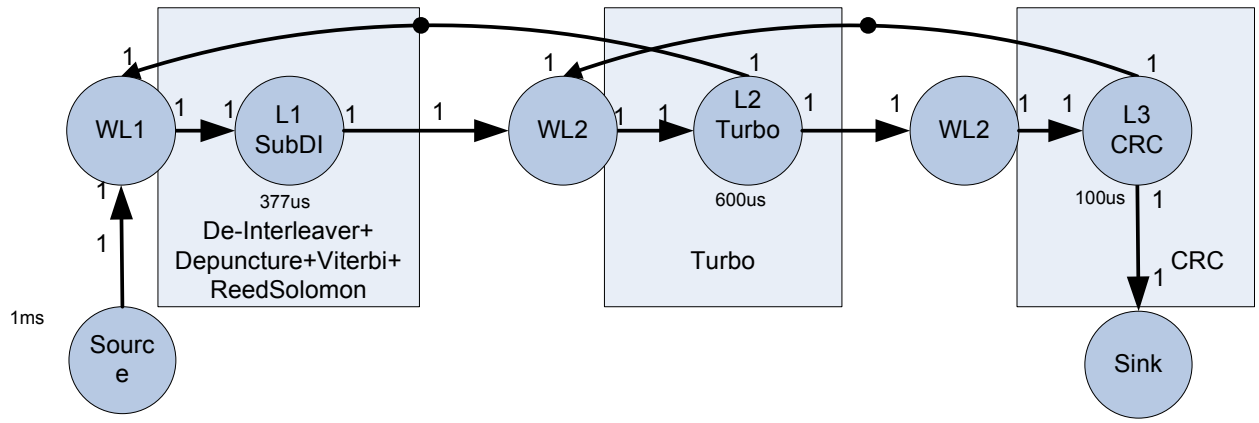

Figure A. 8 Analysis model for LTE without coupled scheduling

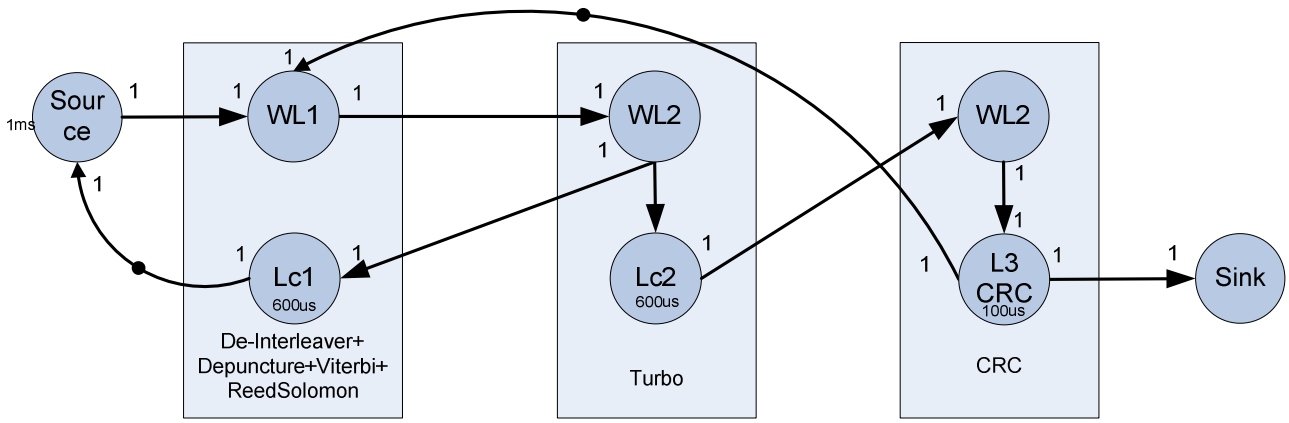

Figure A. 9 Analysis model for LTE with coupled scheduling

A.5 Coware task graph

For DVB-SH, please see page 47.

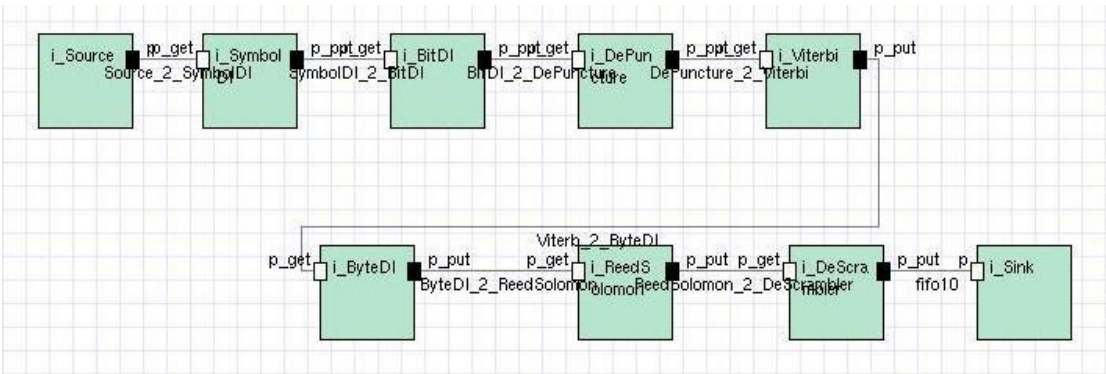

Figure A. 10 Coware task graph for DVB-T

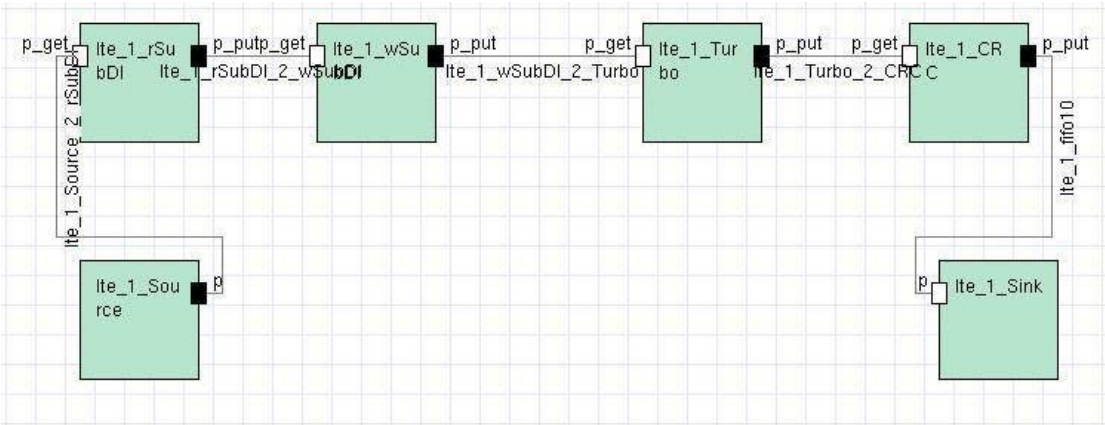

Figure A. 11 Coware task graph for LTE